



z/VM

CMS Pipelines User's Guide and Reference

Version 7 Release 1

Note:

Before using this information and the product it supports, read the information in “Notices” on page 949.

This edition applies to *CMS Pipelines* Version 1 Release 1 Modification Level 12 sublevel 12 and to all subsequent releases and modifications of this product until otherwise indicated in new editions.

! This edition replaces SC24-6252-00

Changes or additions to the text and illustrations are indicated as described in “Summary of Changes” on page xxviii.

© **Copyright International Business Machines Corporation 1986, 2020.**

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

Contents

About This Book	xviii
What Is <i>CMS Pipelines</i> ?	xviii
Who Is <i>CMS Pipelines</i> for?	xix
Skills Expected	xix
How to Use this Book	xix
When Viewing this Book with a PDF Viewer	xx
Web Links	xx
Additional Information, Download Site	xx
Syntax Notation and Typography	xxi
Examples	xxi
Stage Separator	xxi
Supported Operating Environments	xxi
VM Environment	xxii
z/OS Environment	xxii
Compatibility with Older Releases	xxii
Migrating from Older Releases	xxiv
General Compatibility Concerns	xxiv
z/VM 6.3	xxv
Runtime Library Distribution	xxv
Significant Documentation Fixes	xxvi
How to Send Your Comments to IBM	xxvii
Summary of Changes	xxviii
SC24-6252-01, <i>CMS Pipelines</i> 1.1.12/0012	xxviii

Part 1. Introduction 1

Chapter 1. Summary and Two Examples	2
Summary	2
Many Streams	2
Writing Programs	3
A Pipeline Example	3
Another Example	5
Chapter 2. A Walk Through a Pipeline	7
Getting Data In and Out of the Pipeline	7
Filtering Pipeline Data	9
Subroutine Pipelines	10
Writing EXECs with Pipeline Commands	13
Writing a REXX Program to Process Data in the Pipeline	15
Issuing CMS Commands	17
Multistream Pipelines	18
A DB2 Query	20

Part 2. Task Oriented Guide 21

Chapter 3. Where Do I Start?	22
IBM Manuals	22

Contents

Tutorials and Papers	22
Ensure <i>CMS Pipelines</i> Is Installed	22
Find the Stage Separator on Your Terminal	23
TSO Logon Procedure	23
Pipe Help	23
Editing Tools	24
Using FMTP	24
Using SCM	26
Issuing the PIPE Command from a FILELIST Panel	27
Sample Pipelines and REXX Filters	27
Compatibility Between <i>TSO Pipelines</i> and <i>CMS Pipelines</i>	27
Chapter 4. Building a PIPE Command	29
Using Device Drivers to Get Data in and out of a Pipeline	29
Reading and Writing CMS Files	29
Reading and Writing MVS Files	30
OpenExtensions Text Files	31
Libraries	32
Typing on the Terminal	33
Injecting Data into the Pipeline	34
Console Stack (External Data Queue)	34
Using Virtual Unit Record Devices (VM/CMS)	34
MVS SPOOL	36
Accessing Variables	36
Using Device Drivers to Read Data into the Pipeline Downstream	37
Another Way to Read a File	38
Issuing Commands	39
CP	40
CMS	41
TSO	41
Subcommand Environments	42
Obtaining CP Messages and other Console Output	43
Using Filters	44
Translate Characters	44
Counting	46
Editing and Conversion	47
Specifying Input Ranges	50
Selecting Records	51
Splitting, Chopping, and Stripping	57
Joining	60
Changing Record Formats	61
Sorting	62
Cascading Filters	63
Netdata Format	65
IEBCOPY Unloaded Data	66
Building a Selection Key	66
Selecting, Revisited	67
Obtaining Information about Files	70
CMS files	70
TSO data sets	72
Chapter 5. Using Multistream Pipelines	74
Building Blocks for Multistream Pipelines	76
Combining Data Streams	78

	Splitting a Data Stream	79
	Generating a CMS Macro Library	80
	Decoding Trees	82
	Remembering Past Data	83
	Destructive Testing	84
	Other Multistream Programs	86
	Update	86
	Merge	87
	Collate	87
	Lookup	87
	Some Fine Points	88
	Ensure the Pipeline Does not Stall	88
	Keep the Order of Records	89
	Allow End-of-file to Travel Backwards	90
	Chapter 6. Processing Structured Data	91
:	Defining Structured Data	91
:	Activating a Structure Definition	92
:	Referencing Fields in a Structure	93
:	Using Typed Data	93
:	Using Arrays	94
:	Deactivating a Structure Definition	94
:	Structure Scopes	95
:	Caller Scope	95
:	Set Scope	95
:	Thread Scope	96
:	Built-in Scope	96
	Chapter 7. Writing a REXX Program to Run in a Pipeline	97
	Reading and Writing the Pipeline	97
	Using Multiple Streams in REXX Filters	101
	Controlling Streams	102
	Using CALLPIPE to Run a Subroutine Pipeline	103
	Sipping at Data—Processing the Input File Piecemeal	104
	Short Circuits	104
	Accessing REXX Variables	105
	Obtaining the Source String	105
	Scanning the Argument String	105
	Getting a Range from an Input Record	106
	Building Production Strength REXX Filters	107
	Scanning Arguments	107
	Issuing Error Messages	108
	Using the COMMIT Pipeline Command to Ensure other Stages Are Committed to Process Data	109
	Propagating End-of-file	110
	A Complete Robust REXX Filter	111
	Building a REXX Program Dynamically	113
	Implementing a REXX Macro Processor	113
	Miscellaneous Issues	114
	Issuing Commands from a REXX Filter on CMS	114
	Issuing Commands from a REXX Filter on TSO	115
	Issuing Pipeline Commands from an External Function	116
	Return Codes -3 and -7	116
	Pitfalls	116

Contents

Calling External Functions from a REXX Filter	117
The Dangers of Using Implied REXX Filters	117
Performance	118
Should You Compile Your REXX Filters?	118
MVS Considerations	119
Chapter 8. Using Pipeline Options	120
Options for the Pipeline Specification Parser	120
Options for the Pipeline Dispatcher	122
Chapter 9. Debugging	124
Error Messages	124
Other Hints	124
Who Did That?	125
No Output	126
Pipeline Stall	126
Chapter 10. Pipeline Idioms—or—Frequently Asked Questions	127
How Can I Do xxx and Get the Result into REXX Variables?	127
Locating One of Several Targets	128
Making Things Case Insensitive	128
Numeric Sorting	128
Hexadecimal Sorting	131
Obtaining the Length of Records	131
Running a Filter on Part of the Record	132
When the Sort Does Not	133
Why Does QUERY CMSTYPE not Work?	133
Why Does SPLIT 80 Not Work?	133
Why Can't I Update a Stemmed Array?	134
Wondering If It Is a Bug?	134

Part 3. Specialised Topics, Tutorials 137

Chapter 11. Accessing and Maintaining Relational Databases (DB2 Tables)	138
<i>sqlselect</i> —Format a Query	138
Creating, Loading, and Querying a Table	139
Using <i>spec</i> to Convert Fields	142
About the Unit of Work	142
Using Multiple Streams with <i>sql</i> Stages	143
Using Concurrent <i>sql</i> Stages	143
CMS Considerations	143
Obtaining Help	143
Chapter 12. Using CMS Pipelines with Interactive System Productivity Facility	145
Issuing ISPF Commands from REXX Filters	145
Accessing ISPF Tables	145
Accessing ISPF Function Pool Variables	147
Interaction (on TSO) Between ISPF and Stages that Access REXX Variables	148
Defining PIPE to ISPF	148
Chapter 13. SPOOL Files and Virtual SPOOL Devices on VM	149
Introduction to Unit Record Equipment	149
VM SPOOL Files Contain More than Just Cards	151

	Overview of Unit Record Device Drivers	152
	Creating a SPOOL File	152
	Errors on Unit Record Output Devices	153
	Controlling a Unit Record Output Device	154
	Reader SPOOL Files	154
:	Chapter 14. Using VMCF with CMS Pipelines	156
:	Supported Functions	156
:	Identify	156
:	Sendx	156
:	Send	156
:	Send/receive	157
:	Parameter lists	157
:	Example Server Application	158
	Chapter 15. Event-driven Pipelines in Clients and Servers	160
	Waking Up Once a Minute	160
	Terminating an Event-driven Pipeline	161
	Reacting to Immediate Commands	162
	Processing Messages	163
	Validating a User ID	165
	Chapter 16. spec Tutorial	166
	Basic Mechanics	166
	Basic Field Handling	167
	Input Ranges	167
	Literals	171
:	Manifest Constants	172
	The Record Number	173
	Output Placement	173
	Padding	174
	Conversion	175
	Combining Input Records into One Output Record	176
	Multiple Input Streams	176
	Generating Several Output Records from One Input Record	177
	Multiple Output Streams	178
	Expressions	178
	Counter Expressions	178
:	String Processing	181
:	Dealing with Errors in Expressions	182
	Special Processing at End-of-file	183
	Pictures	185
	Boolean Operators	190
	Conditional Processing	192
	The Second Reading Station	194
	Control Breaks	195
	Suppressing Repetitions	196
	Generating Title Records	196
	Printing Subtotals	197
	Break Hierarchies	200
	When <i>spec</i> Establishes a Break	201
:	Suppressing Detail Printing	203
	Driving <i>spec</i> with Due Care and Attention	204
	Examples	205

Contents

Page Formatter	205
And Finally	207
Chapter 17. Rita, the CMS Pipelines Runtime Profiler	208
Example	209
Chapter 18. Using VM Data Spaces with CMS Pipelines	210
Terminology	210
Querying an Address Space	211
Accessing the Contents of a Data Space	212
Creating a Data Space	213
Sharing Address Spaces	214
Using Mapped Minidisks	215
Destroying a Data Space	219
Chapter 19. CMS Pipelines Built-in Programs supporting Data Spaces	220
<hr/>	
Part 4. Reference	221
Chapter 20. Syntax Notation	222
How to Read a Syntax Diagram	222
Syntactic Variables	223
Input Range	228
CMS File Names	231
Mixed case File Names	231
File Mode *	232
Shared File System Considerations	232
MVS File Names	233
OpenExtensions File Names	236
Chapter 21. Syntax of a Pipeline Specification Used with PIPE, runpipe, ADDPIPE, and CALLPIPE	237
Options	237
Pipeline	239
Stage	240
Connectors	241
Labels	242
Example	242
Considerations when Issuing the PIPE Command	243
REXX Limit of 500 Characters in Clause	243
Pipelines in XEDIT Macros	244
Chapter 22. Scanning a Pipeline Specification and Running Pipeline Programs	245
Pipeline Scanner	245
Pipeline Dispatcher	245
States of a Stage	246
Commit Level	247
Reading, Writing	249
Delaying the Record	250
Device Drivers that Wait for External Events	251
Return Codes	252
Chapter 23. Inventory of Built-in Programs	253

Overview by Category	254
<—Read a File	263
< <i>m</i> <i>dsk</i> —Read a CMS File from a Mode	264
< <i>m</i> <i>v</i> <i>s</i> —Read a Physical Sequential Data Set or a Member of a Partitioned Data Set	265
< <i>o</i> <i>e</i> —Read an OpenExtensions Text File	266
< <i>s</i> <i>f</i> <i>s</i> —Read an SFS File	267
< <i>s</i> <i>f</i> <i>s</i> <i>s</i> <i>l</i> <i>o</i> <i>w</i> —Read an SFS File	268
>—Replace or Create a File	270
> <i>m</i> <i>dsk</i> —Replace or Create a CMS File on a Mode	271
> <i>m</i> <i>v</i> <i>s</i> —Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set	273
> <i>o</i> <i>e</i> —Replace or Create an OpenExtensions Text File	275
> <i>s</i> <i>f</i> <i>s</i> —Replace or Create an SFS File	276
>>—Append to or Create a File	278
>> <i>m</i> <i>dsk</i> —Append to or Create a CMS File on a Mode	279
>> <i>m</i> <i>v</i> <i>s</i> —Append to a Physical Sequential Data Set	281
>> <i>o</i> <i>e</i> —Append to or Create an OpenExtensions Text File	282
>> <i>s</i> <i>f</i> <i>s</i> —Append to or Create an SFS File	282
>> <i>s</i> <i>f</i> <i>s</i> <i>s</i> <i>l</i> <i>o</i> <i>w</i> —Append to or Create an SFS File	285
<i>abbrev</i> —Select Records that Contain an Abbreviation of a Word in the First Positions	287
<i>acigroup</i> —Write ACI Group for Users	288
<i>addrdw</i> —Prefix Record Descriptor Word to Records	289
<i>adrspac</i> —Manage Address Spaces	290
<i>aftfst</i> —Write Information about Open Files	293
<i>aggrc</i> —Compute Aggregate Return Code	294
<i>all</i> —Select Lines Containing Strings (or Not)	295
<i>alserv</i> —Manage the Virtual Machine’s Access List	296
<i>apldecode</i> —Process Graphic Escape Sequences	298
<i>aplencode</i> —Generate Graphic Escape Sequences	299
<i>append</i> —Put Output from a Device Driver after Data on the Primary Input Stream	300
<i>asatmc</i> —Convert ASA Carriage Control to CCW Operation Codes	302
<i>asmcont</i> —Join Multiline Assembler Statements	303
<i>asmfind</i> —Select Statements from an Assembler File as XEDIT Find	304
<i>asmnfind</i> —Select Statements from an Assembler File as XEDIT NFind	306
<i>asmxpnd</i> —Expand Joined Assembler Statements	307
<i>beat</i> —Mark when Records Do not Arrive within Interval	308
<i>between</i> —Select Records Between Labels	309
<i>block</i> —Block to an External Format	310
<i>browse</i> —Display Data on a 3270 Terminal	315
<i>buffer</i> —Buffer Records	317
<i>buildscr</i> —Build a 3270 Data Stream	319
<i>casei</i> —Run Selection Stage in Case Insensitive Manner	322
<i>change</i> —Substitute Contents of Records	323
<i>chop</i> —Truncate the Record	326
<i>cipher</i> —Encrypt and Decrypt Using a Block Cipher	328
<i>ckddebloc</i> —Deblock Track Data Record	330
<i>cms</i> —Issue CMS Commands, Write Response to Pipeline	331
<i>collate</i> —Collate Streams	332
<i>combine</i> —Combine Data from a Run of Records	335
<i>command</i> —Issue CMS Commands, Write Response to Pipeline	337
<i>command</i> —Issue TSO Commands	339
<i>configure</i> —Set and Query <i>CMS Pipelines</i> Configuration Variables	340
<i>console</i> —Read or Write the Terminal in Line Mode	341

Contents

<i>copy</i> —Copy Records, Allowing for a One Record Delay	343
<i>count</i> —Count Lines, Blank-delimited Words, and Bytes	344
<i>cp</i> —Issue CP Commands, Write Response to Pipeline	345
<i>crc</i> —Compute Cyclic Redundancy Code	347
<i>c14to38</i> —Combine Overstruck Characters to Single Code Point	350
<i>dam</i> —Pass Records Once Primed	351
<i>dateconvert</i> —Convert Date Formats	352
<i>deal</i> —Pass Input Records to Output Streams Round Robin	360
<i>deblock</i> —Deblock External Data Formats	365
<i>delay</i> —Suspend Stream	369
<i>devinfo</i> —Write Device Information	371
<i>dfsrt</i> —Interface to DFSORT/CMS	372
<i>diag4</i> —Submit Diagnose E4 Requests	373
<i>digest</i> —Compute a Message Digest	374
<i>diskback</i> —Read a File Backwards	376
<i>diskfast</i> —Read, Create, or Append to a File	376
<i>diskid</i> —Map CMS Reserved Minidisk	378
<i>diskrandom</i> —Random Access a File	378
<i>diskslow</i> —Read, Create, or Append to a File	379
<i>diskupdate</i> —Replace Records in a File	380
<i>drop</i> —Discard Records from the Beginning or the End of the File	381
<i>duplicate</i> —Copy Records	382
<i>elastic</i> —Buffer Sufficient Records to Prevent Stall	384
<i>emsg</i> —Issue Messages	385
<i>eofback</i> —Run an Output Device Driver and Propagate End-of-file Backwards	386
<i>escape</i> —Insert Escape Characters in the Record	387
<i>fanin</i> —Concatenate Streams	387
<i>faninany</i> —Copy Records from Whichever Input Stream Has One	388
<i>fanintwo</i> —Pass Records to Primary Output Stream	389
<i>fanout</i> —Copy Records from the Primary Input Stream to All Output Streams	391
<i>fanoutwo</i> —Copy Records from the Primary Input Stream to Both Output Streams	392
<i>fbaread</i> —Read Blocks from a Fixed Block Architecture Drive	393
<i>fbawrite</i> —Write Blocks to a Fixed Block Architecture Drive	394
<i>fblock</i> —Block Data, Spanning Input Records	395
<i>filedescriptor</i> —Read or Write an OpenExtensions File that Is Already Open	396
<i>filetoken</i> —Read or Write an SFS File That is Already Open	397
<i>fillup</i> —Pass Records To Output Streams	399
<i>filterpack</i> —Manage Filter Packages	400
<i>find</i> —Select Lines by XEDIT Find Logic	402
<i>fitting</i> —Source or Sink for Copipe Data	404
<i>fntfst</i> —Format a File Status Table (FST) Entry	404
<i>fntlabel</i> —Select Records from the First One with Leading String	406
<i>frtarget</i> —Select Records from the First One Selected by Argument Stage	407
<i>ftp</i> —Connect to an FTP Server and Exchange Data	408
<i>fullscr</i> —Full screen 3270 Write and Read to the Console or Dialed/Attached Screen	413
<i>fullscrq</i> —Write 3270 Device Characteristics	418
<i>fullscrs</i> —Format 3270 Device Characteristics	419
<i>gate</i> —Pass Records Until Stopped	422
<i>gather</i> —Copy Records From Input Streams	423
<i>getfiles</i> —Read Files	425
<i>greg2sec</i> —Convert a Gregorian Timestamp to Second Since Epoch	426
<i>help</i> —Display Help for <i>CMS Pipelines</i> or <i>DB2</i>	427
<i>hfs</i> —Read or Append File in the Hierarchical File System	430

<i>hfsdirectory</i> —Read Contents of a Directory in a Hierarchical File System	431
<i>hfsquery</i> —Write Information Obtained from OpenExtensions into the Pipeline . . .	432
<i>hfsreplace</i> —Replace the Contents of a File in the Hierarchical File System	433
<i>hfsstate</i> —Obtain Information about Files in the Hierarchical File System	434
<i>hfsxecute</i> —Issue OpenExtensions Requests	435
<i>hiasm</i> —Interface to High Level Assembler	437
<i>hiasmerr</i> —Extract Assembler Error Messages from the SYSADATA File	439
<i>hole</i> —Destroy Data	440
<i>hostbyaddr</i> —Resolve IP Address into Domain and Host Name	441
<i>hostbyname</i> —Resolve a Domain Name into an IP Address	442
<i>hostid</i> —Write TCP/IP Default IP Address	443
<i>hostname</i> —Write TCP/IP Host Name	444
<i>httpsplit</i> —Split HTTP Data Stream	445
<i>iebcopy</i> —Process IEBCOPY Data Format	446
<i>if</i> —Process Records Conditionally	447
<i>imcmd</i> —Write the Argument String from Immediate Commands	448
<i>insert</i> —Insert String in Records	450
<i>inside</i> —Select Records between Labels	450
<i>instore</i> —Load the File into a storage Buffer	451
<i>ip2socka</i> —Build sockaddr_in Structure	454
<i>ispf</i> —Access ISPF Tables	455
<i>jeremy</i> —Write Pipeline Status to the Pipeline	457
<i>join</i> —Join Records	458
<i>joincont</i> —Join Continuation Lines	460
<i>juxtapose</i> —Preface Record with Marker	462
<i>ldrtbls</i> —Resolve a Name from the CMS Loader Tables	464
<i>listcat</i> —Obtain Data Set Names	465
<i>listdsi</i> —Obtain Information about Data Sets	466
<i>listispcf</i> —Read Directory of a Partitioned Data Set into the Pipeline	468
<i>listpds</i> —Read Directory of a Partitioned Data Set into the Pipeline	469
<i>literal</i> —Write the Argument String	471
<i>locate</i> —Select Lines that Contain a String	472
<i>lookup</i> —Find Records in a Reference Using a Key Field	474
<i>maclib</i> —Generate a Macro Library from Stacked Members in a COPY File	483
<i>mapdisk</i> —Map Minidisks Into Data spaces	484
<i>mctoasa</i> —Convert CCW Operation Codes to ASA Carriage Control	486
<i>mdiskblk</i> —Read or Write Minidisk Blocks	487
<i>mdskfast</i> —Read, Create, or Append to a CMS File on a Mode	488
<i>mdskback</i> —Read a CMS File from a Mode Backwards	490
<i>mdskrandom</i> —Random Access a CMS File on a Mode	491
<i>mdskslow</i> —Read, Append to, or Create a CMS File on a Mode	493
<i>mdskupdate</i> —Replace Records in a File on a Mode	495
<i>members</i> —Extract Members from a Partitioned Data Set	496
<i>merge</i> —Merge Streams	498
<i>mqsc</i> —Issue Commands to a WebSphere MQ Queue Manager	499
<i>nfind</i> —Select Lines by XEDIT NFind Logic	500
<i>nlocate</i> —Select Lines that Do Not Contain a String	501
<i>noeofback</i> —Pass Records and Ignore End-of-file on Output	503
<i>not</i> —Run Stage with Output Streams Inverted	503
<i>notinside</i> —Select Records Not between Labels	505
<i>nucext</i> —Call a Nucleus Extension	506
<i>optcdj</i> —Generate Table Reference Character (TRC)	507
<i>outside</i> —Select Records Not between Labels	508
<i>outstore</i> —Unload a File from a storage Buffer	509

Contents

<i>overlay</i> —Overlay Data from Input Streams	510
<i>overstr</i> —Process Overstruck Lines	511
<i>pack</i> —Pack Records as Done by XEDIT and COPYFILE	512
<i>pad</i> —Expand Short Records	514
<i>parcel</i> —Parcel Input Stream Into Records	515
<i>pause</i> —Signal a Pause Event	516
<i>pdsdirect</i> —Write Directory Information from a CMS Simulated Partitioned Data Set	517
<i>pick</i> —Select Lines that Satisfy a Relation	518
<i>pipcmd</i> —Issue Pipeline Commands	523
<i>pipestop</i> —Terminate Stages Waiting for an External Event	525
<i>polish</i> —Reverse Polish Expression Parser	525
<i>predselect</i> —Control Destructive Test of Records	531
<i>preface</i> —Put Output from a Device Driver before Data on the Primary Input Stream	532
<i>printmc</i> —Print Lines	534
<i>punch</i> —Punch Cards	536
<i>qdecode</i> —Decode to Quoted-printable Format	537
<i>qencode</i> —Encode to Quoted-printable Format	538
<i>qsam</i> —Read or Write Physical Sequential Data Set through a DCB	539
<i>query</i> —Query <i>CMS Pipelines</i>	540
<i>random</i> —Generate Pseudorandom Numbers	541
<i>reader</i> —Read from a Virtual Card Reader	542
<i>readpds</i> —Read Members from a Partitioned Data Set	544
<i>retab</i> —Replace Runs of Blanks with Tabulate Characters	545
<i>reverse</i> —Reverse Contents of Records	546
<i>rexx</i> —Run a REXX Program to Process Data	546
<i>rexxvars</i> —Retrieve Variables from a REXX or CLIST Variable Pool	549
<i>runpipe</i> —Issue Pipelines, Intercepting Messages	553
<i>scm</i> —Align REXX Comments	555
<i>sec2greg</i> —Convert Seconds Since Epoch to Gregorian Timestamp	556
<i>sfsback</i> —Read an SFS File Backwards	557
<i>sfsdirectory</i> —List Files in an SFS Directory	559
<i>sfsrandom</i> —Random Access an SFS File	560
<i>sfsupdate</i> —Replace Records in an SFS File	562
<i>snake</i> —Build Multicolumn Page Layout	565
<i>socka2ip</i> —Format sockaddr_in Structure	566
<i>sort</i> —Order Records	567
<i>space</i> —Space Words Like REXX	569
<i>spec</i> —Rearrange Contents of Records	571
<i>spill</i> —Spill Long Lines at Word Boundaries	577
<i>split</i> —Split Records Relative to a Target	580
<i>sql</i> —Interface to SQL	582
<i>sqlcodes</i> —Write the last 11 SQL Codes Received	587
<i>sqlselect</i> —Query a Database and Format Result	587
<i>stack</i> —Read or Write the Program Stack	588
<i>starmon</i> —Write Records from the *MONITOR System Service	589
<i>starmsg</i> —Write Lines from a CP System Service	591
<i>starsys</i> —Write Lines from a Two-way CP System Service	594
<i>state</i> —Provide Information about CMS Files	597
<i>state</i> —Verify that Data Set Exists	599
<i>statew</i> —Provide Information about Writable CMS Files	600
<i>stem</i> —Retrieve or Set Variables in a REXX or CLIST Variable Pool	603
<i>stfle</i> —Store Facilities List	606

<i>storage</i> —Read or Write Virtual Machine Storage	607
<i>strasmfind</i> —Select Statements from an Assembler File as XEDIT Find	609
<i>strasmnfind</i> —Select Statements from an Assembler File as XEDIT NFind	610
<i>strfind</i> —Select Lines by XEDIT Find Logic	611
<i>strfrlabel</i> —Select Records from the First One with Leading String	612
<i>strip</i> —Remove Leading or Trailing Characters	613
<i>strliteral</i> —Write the Argument String	614
<i>strnfind</i> —Select Lines by XEDIT NFind Logic	616
<i>strtolabel</i> —Select Records to the First One with Leading String	617
<i>structure</i> —Manage Structure Definitions	618
<i>strwhilelabel</i> —Select Run of Records with Leading String	625
<i>stsi</i> —Store System Information	626
<i>subcom</i> —Issue Commands to a Subcommand Environment	626
<i>substring</i> —Write substring of record	628
<i>synchronise</i> —Synchronise Records on Multiple Streams	628
<i>sysdsn</i> —Test whether Data Set Exists	630
<i>sysout</i> —Write System Output Data Set	631
<i>sysvar</i> —Write System Variables to the Pipeline	632
<i>take</i> —Select Records from the Beginning or End of the File	633
<i>tape</i> —Read or Write Tapes	634
<i>tcpcksum</i> —Compute One’s complement Checksum of a Message	637
<i>tcpclient</i> —Connect to a TCP/IP Server and Exchange Data	638
<i>tcpdata</i> —Read from and Write to a TCP/IP Socket	643
<i>tcplisten</i> —Listen on a TCP Port	648
<i>threeway</i> —Split record three ways	651
<i>timestamp</i> —Prefix the Date and Time to Records	652
<i>tokenise</i> —Tokenise Records	654
<i>tolabel</i> —Select Records to the First One with Leading String	655
<i>totarget</i> —Select Records to the First One Selected by Argument Stage	656
<i>trackblock</i> —Build Track Record	657
<i>trackdeblock</i> —Deblock Track	658
<i>trackread</i> —Read Full Tracks from ECKD Device	659
<i>tracksquish</i> —Squish Tracks	660
<i>trackverify</i> —Verify Track Format	660
<i>trackwrite</i> —Write Full Tracks to ECKD Device	661
<i>trackxpan</i> —Unsquish Tracks	662
<i>trfread</i> —Read a Trace File	663
<i>tso</i> —Issue TSO Commands, Write Response to Pipeline	664
<i>udp</i> —Read and Write an UDP Port	665
<i>unique</i> —Discard or Retain Duplicate Lines	668
<i>unpack</i> —Unpack a Packed File	670
<i>untab</i> —Replace Tabulate Characters with Blanks	671
<i>update</i> —Apply an Update File	672
<i>urldeblock</i> —Process Universal Resource Locator	673
<i>uro</i> —Write Unit Record Output	674
<i>utf</i> —Convert between UTF-8, UTF-16, and UTF-32	676
<i>var</i> —Retrieve or Set a Variable in a REXX or CLIST Variable Pool	678
<i>vardrop</i> —Drop Variables in a REXX Variable Pool	681
<i>varfetch</i> —Fetch Variables in a REXX or CLIST Variable Pool	683
<i>varload</i> —Set Variables in a REXX or CLIST Variable Pool	685
<i>varset</i> —Set Variables in a REXX or CLIST Variable Pool	688
<i>vchar</i> —Recode Characters to Different Length	690
<i>verify</i> —Verify that Record Contains only Specified Characters	692
<i>vmc</i> —Write VMCF Reply	693

Contents

:	<i>vmcdata</i> —Receive, Reply, or Reject a Send or Send/receive Request	694
:	<i>vmclient</i> —Send VMCF Requests	695
:	<i>vmclisten</i> —Listen for VMCF Requests	696
:	<i>waitdev</i> —Wait for an Interrupt from a Device	697
:	<i>warp</i> —Pipeline Wormhole	698
:	<i>warplist</i> —List Wormholes	699
:	<i>whilelabel</i> —Select Run of Records with Leading String	700
:	<i>wildcard</i> —Select Records Matching a Pattern	701
:	<i>writepds</i> —Store Members into a Partitioned Data Set	703
:	<i>xab</i> —Read or Write External Attribute Buffers	705
:	<i>xedit</i> —Read or Write a File in the XEDIT Ring	705
:	<i>xlate</i> —Transliterate Contents of Records	708
:	<i>xmsg</i> —Issue XEDIT Messages	712
:	<i>xpndhi</i> —Expand Highlighting to Space between Words	713
:	<i>xrange</i> —Write a Range of Characters	713
:	<i>zone</i> —Run Selection Stage on Subset of Input Record	714
:	<i>3277bfra</i> —Convert a 3270 Buffer Address Between Representations	715
:	<i>3277enc</i> —Write the 3277 6-bit Encoding Vector	717
:	<i>64decode</i> —Decode MIME Base-64 Format	717
:	<i>64encode</i> —Encode to MIME Base-64 Format	718
	Chapter 24. <i>spec</i> Reference	719
	Overview	719
	Concepts	720
	The Cycle	720
	Streams	720
	Field Identifiers, Control Breaks, Break Levels	720
:	Structured Data	721
	Counters	722
	Number Representation	722
	Expressions	723
	Syntax Recursion	723
	Syntax Description	723
	Syntax Overview	723
	Main Options	723
	Item Group	724
	If Group	725
:	While Group	725
	Plain Item	726
	Stream Control	727
	Break Control	727
	Data Field	728
	Input Source	729
	Conversions	731
	Output Placement	734
	Expression	736
	Assignment Expression	736
	Conditional Expression	737
	Binary Expression	738
	Term	739
	Floating point Numbers	741
	Functions	741
	Pictures	747
	Sign Characters	747

Digit Selection	748
Punctuation	748
Implied Decimal Point	748
Exponent	748
General	748
Continental European Conventions	749
Chapter 25. Pipeline Commands	750
ADDPIPE—Add a Pipeline Specification to the Running Set	751
ADDSTREAM—Create a Stream	752
BEGOUTPUT—Enter Implied Output Mode	752
CALLPIPE—Run a Subroutine Pipeline	753
COMMIT—Commit Stage to a New Level	754
EOFREPORT—Enable Reporting of Stream Events	755
GETRANGE—Extract Part of Record or String	756
ISSUEMSG—Issue a Message from the Repository	757
MAXSTREAM—Return the Highest Stream Number	758
MESSAGE—Issue a Message	759
NOCOMMIT—Disable Automatic Commit on I/O	759
OUTPUT—Write a Line	760
PEEKTO—Preview the next Input Line	761
READTO—Read or Discard an Input Line	762
RESOLVE—Return Entry Point of Built-in Program	763
REXXCMD—Call a REXX Pipeline Program from a Filter	763
SCANRANGE—Parse an input range	764
SCANSTRING—Parse a delimited string	766
SELECT—Select a Stream	766
SETRC—Set Return Code in Stage Writing	767
SEVER—Break a Connection	768
SHORT—Connect Input and Output Stream	769
STAGENUM—Return Stage’s Position in Pipeline	769
STREAMNUM—Return Stream Number	770
STREAMSTATE—Return Stream Status	770
SUSPEND—Allow other Stages to Run	772
Chapter 26. Message Reference	773
Chapter 27. PIPMOD Command (CMS Pipelines only)	863
The PIPE Bootstrap Module	863
The PIPMOD Nucleus Extension	863
Setting Permanent Pipeline Options	864
The Message Level	864
PIPMOD Immediate Commands	865
ACTIVE—Show the Active Stage	866
STOP—Terminating Stages that Wait Forever	866
WHERE—Show Addresses of Pipeline Control Blocks	866
Chapter 28. Configuring CMS Pipelines	867
Default Styles	867
CMS Considerations	867
Configuration Variables	868
Diskreplace	868
Disktempfiletype	868
Group	869

Contents

	Repository	869
	SQLpgmname	869
	SQLpgmowner	869
	Stallaction	870
	Stallfiletype	870
	Style	870
	Installation-wide Customisation (CMS)	871
:	Chapter 29. Diagnosis	872
:	Determining and Terminating the Currently Running Stage	872
:	VM	872
:	Traps	873
<hr/>		
	Part 5. Appendices	875
	Appendix A. Summary of Built-in Programs	876
	Appendix B. Messages, Sorted by Text	897
	Appendix C. Implementing CMS Commands as Stages in a Pipeline	918
	Appendix D. Running Multiple Versions of <i>CMS Pipelines</i> Concurrently	921
	Basic Initialisation	921
	Initialisation of a Shared Segment	921
	Coexistence	922
!	Filter Packages	922
:	Appendix E. Generating and Using Filter Packages with <i>CMS Pipelines</i>	924
:	Note for MVS Users	924
:	Introduction	924
:	Specifying Files	925
:	Contents of a Filter Package	926
:	Glue Code	926
:	Entry Point Table	926
:	FPLEPTBL—Generate Entry Point Table Object Module	927
:	Message Text Table	927
:	FPLMSGTB—Generate Message Text Table Object Module	928
:	Keyword Table	929
:	FPLKWDTB—Generate a Keyword Table Object Module	929
!	PIPGFTXT—Generate Object Module from Program Directory	930
!	PIPGFMOD—Generate Filter package Load Module	930
:	Programs	930
:	Generating a Sample Type-1 Filter Package	931
:	Appendix F. Pipeline Compatibility and Portability between CMS and TSO	933
:	TSO Commands Supplied with <i>TSO Pipelines</i>	933
:	FPLRESET	933
:	FPLDEBUG	933
:	FPLUNIX	934
:	Using the PIPE Command from Unix System Services	934
:	Pipeline Specifications—The PIPE Command	934
:	Appendix G. Format of Output Records from <i>runpipe</i> EVENTS	939

Contents

00—Message	939
01—Begin Pipeline Set	940
02—End Pipeline Set	940
03—Enter Scanner	940
04—Pipeline Vector Allocated	940
05—Leave Scanner	942
06—Scanner Item	942
07—Calling Syntax Exit	943
08—Start Stage	944
09—End Stage	944
0A—Resuming Stage	944
0B—Calling Dispatcher Service	945
0C—Pipeline is Stalled	946
0D—State of Stage	946
0E—Pipeline Committing	946
0F—Console Input	947
10—Console Output	947
11—Pause	947
12—Subroutine Pipeline Complete	948
13—Caller is Waiting for Subroutine Pipeline to Complete	948
Notices	949
Programming Interface Information	950
Trademarks	950
Terms and Conditions for Product Documentation	950
IBM Online Privacy Statement	951
Glossary	952
Bibliography	957
Where to Get z/VM Information	957
Additional References	957
Index	959

About This Book

This book has a dual purpose: to introduce new users to *CMS Pipelines*, and to provide reference information for all *CMS Pipelines* users.

Though this book is specific to z/VM, it also describes built-in programs that are specific to z/OS and contains other references to z/OS and *TSO Pipelines*. These references do not imply that an implementation of *CMS Pipelines* for z/OS is available.

What Is *CMS Pipelines*?

CMS Pipelines implements the pipeline concept under the VM/CMS and the z/OS operating systems. Programs running in a pipeline operate on a sequential stream of records that are read and written through a device independent interface. Any program can be combined with any other one because all pipeline programs read and write records through this device independent interface.

CMS Pipelines provides a CMS and TSO command, PIPE. The argument string to the PIPE command is called a pipeline specification. PIPE selects programs and “bolt” them together in a pipeline to pump data through them. The pipeline module has a built-in library of programs that can be called in a pipeline specification; these programs interface to z/OS and CP/CMS, and perform many utility functions. For example, read a file, select particular records, reformat each record, and display the result on the terminal; *CMS Pipelines* takes the chores out of this task because it has utility functions to read files and write to your terminal. It might even have programs to perform the selection and editing you want, but if it does not, all you do is write a program to complement the built-in programs rather than start from scratch.

CMS Pipelines users issue pipeline commands from the terminal or in EXEC procedures; they can write programs in REXX to augment the programs built into *CMS Pipelines*. The PIPE module can also run as a job step in z/OS batch.

Programming a complex algorithm is a matter of selecting building blocks; *CMS Pipelines* fits them together.

The concept of a simple (“straight”) pipeline is extended in these ways:

- A program can define a subroutine pipeline to perform a function on all or part of its input data.
- A network of intersecting pipelines lets a program be in several pipelines concurrently where it has access to multiple data streams.
- A program can dynamically redefine the pipeline topology to replace itself with another pipeline; or to insert a pipeline segment before or after itself, or both.

CMS Pipelines offers several features to improve the robustness of pipelines:

- A syntax error in the overall pipeline structure or in any one program causes the entire pipeline to be suppressed.
- The inability to allocate required resources causes all programs to release what they have allocated and terminate before any irreversible actions are taken.
- Errors while data flow in the pipeline can be detected by all participating programs. For example, a disk file might not be replaced in such circumstances.

Who Is *CMS Pipelines* for?

CMS Pipelines can be useful for all CMS and TSO users; how you may wish to use it may depend on your role:

- **End users** can process data in a way that is not procedural, often called *functional programming*. Data processing professionals often underestimate the mental abilities of their users, thinking that anything different from “normal” programming is difficult. Many end users find “simple programs” exasperating and prefer to manipulate data at a more general level; *CMS Pipelines* provides this capability, with interfaces to ISPF tables and SQL or DB2.
- **Programmers** often find that *CMS Pipelines* helps them write better programs faster and cheaper. Access to files and devices is greatly simplified; programs are reused without change. Of course, *CMS Pipelines* does not make a sloppy programmer better overnight, but *pipethink* makes it easy to break a complex task into smaller ones and thus reduce complexity.
- **Toolsmiths** write tools: programs to help programmers and users be more productive when pursuing their business objectives. Toolsmiths have a field day with *CMS Pipelines*; it takes the toil out of writing CMS and TSO tools. The toolsmith concentrates on the *real* problem instead of, for instance, how to access files most efficiently.

There is an avalanche effect when the toolsmith makes more and better tools for the programmer who, in turn, writes more functions for end users!

Skills Expected

You should be familiar with the timesharing system on which you are going to use *CMS Pipelines*: CMS is described in the *&dmsb3*; TSO is described in *TSO Extensions Version 2 User's Guide*, SC28-1880.

! Some experience with REXX (described in the *z/VM: REXX/VM User's Guide*, SC24-6315
! the *z/VM: REXX/VM Reference*, SC24-6314 and the *TSO Extensions Version 2 REXX Reference*, SC28-1883) is recommended, at least to the point of writing simple command procedures.

Some CMS or TSO skill, but no (systems) programming expertise is required to understand the concepts of *CMS Pipelines* and use most of the built-in programs. Some built-in programs, however, expose operating system interfaces that require an understanding of device architectures or the format of data sent to a device.

TSO users are sometimes expected to perform a mental transformation when an example is explained in CMS terms. This transformation is often a matter of different file naming conventions, or to substitute ISPF/PDF for XEDIT as the editor being used; *CMS Pipelines* is designed to provide a common set of functions for the two timesharing systems.

How to Use this Book

Chapter 1, “Summary and Two Examples” on page 2 gives a quick explanation for the experienced timesharing user.

New users of CMS and *CMS Pipelines* should read Chapter 2, “A Walk Through a Pipeline” on page 7 and progress to the task-oriented guide.

About This Book

Experienced CMS users may find the introduction too slow; go directly to Part 2, “Task Oriented Guide” on page 21 instead.

Having mastered the topics explained in Part 2, you might like to peruse selected chapters of Part 3, “Specialised Topics, Tutorials” on page 137; and you might want to familiarise yourself with the tutorial on *specs*.

Though Part 4 is intended for reference use, you should read the first chapter if you find that the syntax notation is not intuitive. The information in Part 4 is available in the help library, which you can access from your terminal.

When Viewing this Book with a PDF Viewer

This book contains many hyperlinks; in fact, so many that highlighting them in coloured frames would become distracting. Thus, links to other documents and links to the world wide web are the only links that are highlighted.

Intra-book links include references to *CMS Pipelines* terms and concepts. For example, all mention of a built-in program also includes a link. Try this: *console*.

Thus, if you are wondering what something means, try hovering the mouse pointer over it and see whether it offers a link. Be sure to use the hand tool; other tools may not offer the links.

Web Links

The default for Acrobat is to include web links into the document you are viewing (editing, actually), but this is unlikely to be what you want to happen when viewing this document.

Select Edit, Preferences, Web capture and ensure that the “Open Weblinks” drop-down is set to “In Web Browser”.

Additional Information, Download Site

Check out the VM home page:

<http://www.vm.ibm.com>

Also check out the *CMS Pipelines* home page:

<http://vm.marist.edu/%7Epipeline>

%7E represents the tilde character (~); you can use either notation.

Obtain a copy of *CMS Pipelines Tutorial, GG66-3158* from the *CMS Pipelines* homepage. Even though the book is dated, it provides a good introduction for the beginning *CMS Pipelines* user, with exercises and examples.

From this site you can also download the *CMS Pipelines* “Runtime Library”, also known as the “Field Test Version” and the “Princeton Distribution”. Note that this version offers no advantages over the version of *CMS Pipelines* that is shipped with z/VM 6.4, so installations using z/VM 6.4 should not install the test version.

The *CMS Pipelines* discussion list is also hosted by Marist College. To join the list, send mail with a subject line that contains “SUBSCRIBE CMS-PIPELINES” to:

listserv@vm.marist.edu

Syntax Notation and Typography

The syntax diagrams in the inventory of built-in programs are explained in Chapter 20, “Syntax Notation” on page 222.

A reference to a built-in program is written in italics type, for instance *spec*. It is in lower case, even at the beginning of a sentence. A keyword option is set in small capitals: for instance, ANYOF.

A complete command is written in double quotes in Gothic type: for instance, “pipmod msglevel 15”.

Examples

This book contains many example terminal sessions and command procedures. Examples are set in monospaced Gothic type.

The first position of a line of an example terminal session has a character to show whether the line is typed by the user or is a system response; this character is not part of the line you see or write on your terminal. Commands and input lines written on the terminal, by the user, have a blank (space) in the first column; responses have an arrowhead (►). Most CMS examples were done on CMS in line mode; they show the PIPE command in front of the pipeline specification.

Though not identified, 417 of the CMS examples were run while formatting this book. Figure 266 on page 166 shows the version of *CMS Pipelines* used. What you see is what it does, even if an error should slip by the author.

Other examples are written as fragments of REXX programs; you can tell by the comment (*/* comment */*) on the first line. This indicates that the program is written in REXX.

Stage Separator

The solid vertical bar (|) is an important character when writing *CMS Pipelines* commands; it indicates the end of the specification of one program and the beginning of the next. This character is also used as the logical OR operator in REXX and PLI. Not all terminals display the code point (X'4F') as a solid vertical bar; refer to “Find the Stage Separator on Your Terminal” on page 23 for more information.

Supported Operating Environments

CMS Pipelines Version 1 Release 1 Modification Level 12 supports z/VM (VM/CMS) and z/OS environments.

About This Book

VM Environment

CMS Pipelines is supported on z/VM Version 7 Release 1.

sql was developed with SQL/DS Version 1 Release 3.5, IBM Program Number 5748-XXJ; it has been tested with Version 2 Releases 1 and 2, IBM Program Number 5688-004, Version 3 Release 1, IBM Program Number 5688-103, and IBM Database 2 Server for VSE & VM Version 5. *ispf* was developed for Version 2 Release 2 of Interactive System Productivity Facility, IBM Program Number 5664-282.

CMS Multitasking Considerations

CMS Pipelines supports CMS multitasking. Any number of threads may issue PIPE commands concurrently. *CMS Pipelines* itself creates neither processes nor threads, nor does it switch between threads.

When *CMS Pipelines* waits for an external event, it uses thread suspend/resume when it senses that some other application has entered multitasking mode, so as not to lock out such an application.

64-bit CMS (z/CMS) Considerations

CMS Pipelines has been tested with the 64-bit CMS nucleus. You cannot use ALETs in a 64-bit virtual machine because it does not support XC mode.

z/OS Environment

TSO Pipelines 1.1.12 supports z/OS Enterprise Systems Architecture and z/OS with JES2 or JES3.

Level 1.1.9 of *TSO Pipelines* shipped under the name BatchPipeWorks* as part of BatchPipes/MVS and subsequently SmartBatch/MVS, IBM Program Number 5655-A17, but those products are no longer marketed by IBM.

Compatibility with Older Releases

The release of *CMS Pipelines* included in z/VM 6.4 is available to all CMS users and applications. It does not require any additional software to be downloaded or a specific version to be selected.

CMS Pipelines has been shipped as part of VM/ESA, starting with the latter's Version 1 Release 1 Modification level 1. Even so, *CMS Pipelines* maintained its own level, which is independent of the CMS level. Intermediate releases of *CMS Pipelines* have been made available through the Runtime Library Distribution or as Field Test.

In Figure 1, the first column shows the *CMS Pipelines* level and the sublevel when it was considered complete. The second column shows the date when this happened. The third column shows the VM/ESA or z/VM release that picked up this level. The last column shows the pipeline level reported by the DMSPIPE module.

Figure 1. Equivalent levels of CMS Pipelines

<i>CMS Pipelines Level and Sublevel</i>	Date	Version and Release	Level
1.1.6/0057	5 Aug 1991	VM/ESA 1.1.1 and VM/ESA 1.2.0	21010000
1.1.7/0053	22 Apr 1993	VM/ESA 1.2.1	22010000
		VM/ESA 1.1.5 (370 Feature)	15010000
1.1.8/001C	12 Jan 1994	VM/ESA 1.2.2	22020000
1.1.9/0033	14 Jul 1995	VM/ESA 2.1.0	3109001A ¹
		VM/ESA 2.2.0	31090038
1.1.10/001F	19 Nov 1997	VM/ESA 2.3.0	110A0020
1.1.11/0013	15 Jan 2007		
1.1.12/0006	14 Jul 2010		
1.1.12/000C	30 Jul 2015		
1.1.12/000D		z/VM 6.4	110C000D

Prior to *CMS Pipelines* level 1.1.10 and VM/ESA Version 2 Release 3.0, the versions shown in Figure 1 on page xxii are considered equivalent, which means that pipelines written according to the documentation of one of the environments will also work with the other one. There may be undocumented aspects that do not work in equivalent environments; traditionally, *CMS Pipelines* would quietly accept undocumented behaviour that is compatible with some past specification.

Built-in programs that are described in this book are also available in previous releases of VM/ESA and z/VM unless revision codes indicate new function added since 1.1.10. The built-in programs marked as z/OS are not available in CMS, the ones marked CMS are not available in z/OS.

As of *CMS Pipelines* level 1.1.10 and VM/ESA Version 2 Release 3.0 the two versions of *CMS Pipelines* are consolidated. Backwards compatibility issues are handled by configuration variables (see Chapter 28, “Configuring *CMS Pipelines*” on page 867).

Notes:

1. *CMS Pipelines* 1.1.10 was carried forward unchanged into z/VM versions 3, 4, 5, and 6. However, each release of z/VM has increased its sublevel; thus, for example, 5.4 reports level 110A002A.
2. With Version 7 Release 1, CMS has picked up all new function in *CMS Pipelines* and reports the corresponding level. This level of *CMS Pipelines* is very close to the 1.1.12/000C level of the Runtime Library.

¹ Though in reality, this is close to the function in sublevel 33.

About This Book

3. *CMS Pipelines* 1.1.11 and later functionality is not shipped in z/VM releases before z/VM 6.4. Function not present in earlier releases of z/VM is indicated by an exclamation point or an inverted exclamation point as the revision code in this book.
4. You cannot report errors with the Runtime Library (Field Test Version) through normal IBM program support procedures.

Migrating from Older Releases

In general releases of *CMS Pipelines* are “upwardly compatible”. Applications that use *CMS Pipelines* as specified will continue to run without any change to the application. Any incompatible changes are listed in the release specific sections below.

General Compatibility Concerns

It is possible that an application was developed such that it relies on behaviour of *CMS Pipelines* that was unspecified or on a programming error in *CMS Pipelines*. Such a defect in the application might not be noticed until *CMS Pipelines* is upgraded and the programming error is corrected. Even when *CMS Pipelines* releases are considered compatible, normal procedures should be followed to verify that the application works as expected.

New built-in programs

When new built-in programs are introduced in *CMS Pipelines*, applications may be impacted when they rely on an implicit reference to a REXX program with the same name (the built-in program will be used instead of the REXX program). Applications should avoid such conflicts using the *rexx* built-in program to make the reference to the REXX program explicit.

In many cases, naming conflicts show immediately when the application runs using the new level of *CMS Pipelines*. To be alerted earlier or to detect more subtle problems, developers are encouraged to at least review the revision codes in the Table of Contents to spot any potential naming conflicts.

A pipeline can list the existing REXX programs (in this example on the S disk) and use the RESOLVE option of *filterpack* to check for a potential conflict with names of built-in programs. The output is used to guide further inspection of the application.

```
pipe cms LIST * REXX S | substr w1 | filterpack resolve | ...  
... pick w2 == ,builtin, | console  
►ENBASE64 builtin  
►Ready;
```

Note that in this example, *CMS Pipelines* deliberately introduced the new built-in with the same name. The built-in performs the same function without the overhead of a REXX program. By using the same name, applications automatically take advantage of the faster implementation of the built-in program.

Though less common, a similar conflict may exist with filter packages that provide programs with the same name as built-in programs. Developers can use the MODLIST option of *filterpack* to list the contents of a filter package. The list of names can be checked against the names of built-in programs in the same way as shown for REXX programs.

Dispatching Order and End-of-file Propagation

When a multistream pipeline is constructed in such a way that the relative order in which the stages can run is unpredictable, the actual order in which stages run is unspecified. It may change with the next release (as well as with changes in the application or the data). The application should be corrected to avoid a dependency on unspecified behaviour.

Many *CMS Pipelines* programs propagate end-of-file where applicable to make pipeline segments terminate when there is no value in processing more data, as specified with the description of the programs. When the behaviour is not specified, it may change from one release to another.

Messages and Return Codes

The response of *CMS Pipelines* to usage errors is subject to change. The text of the messages may be changed to provide more accurate information about the error. Internal changes in *CMS Pipelines* may cause the message to be issued by another module, which changes part of the message identifier. In some cases enhancements to *CMS Pipelines* may require a different message to be issued and the PIPE command will end with a different return code. Applications should not have a dependency on particular return codes from built-in programs unless specified as function of the program.

z/VM 6.3

In addition to the aspects in “General Compatibility Concerns” on page xxiv the following incompatible changes apply when migrating to z/VM 6.4.

- The granularity of *timestamp* has been increased such that the default 8-byte time stamp will show also the hundredths of a second. In z/VM 6.3 the last two digits were “00”.
- The scope of the STRIP keyword in *spec* has been corrected not to apply to other references to the same input field.

The online documentation shown with HELP PIPE and *ahelp* has been upgraded to match this publication. The built-ins *fullscr*, *spec* and *starmon* are now referred to in the documentation by their popular abbreviation. The longer name remains an alias.

Runtime Library Distribution

CMS Pipelines included in z/VM 6.4 is functionally equivalent to the Runtime Library Distribution level 1.1.12/000C (2015-07-30). In addition to the aspects in “General Compatibility Concerns” on page xxiv the following incompatible changes apply when migrating to z/VM 6.4.

- The STRING option in *state* and *statew* (added in 1.1.11/0015) has been removed. Where necessary use NOFORMAT on *state* or *statew* followed by a separate *fmtfst* stage to format the date and time as desired.
- The *md5* program has been superseded by *digest* with the MD5 option and will be removed in a future release.
- The default style for *CMS Pipelines* as shipped with z/VM is DMS while the Runtime Library uses FPL. Users are encouraged to review Chapter 28, “Configuring *CMS Pipelines*” on page 867 to understand the differences between the two styles and the options to change the behaviour where necessary.
- With the DMS style, the *help* stage invokes HELP PIPE. The online documentation shown with HELP PIPE and *ahelp* has been upgraded to match this publication. The

About This Book

! built-ins *fullscr*, *spec* and *starmon* are now referred to in the documentation by their
! popular abbreviation. The longer name remains valid as an alias.
!
! Users of older levels of the Runtime Library Distribution should also refer to PIPELINE
! NEWS on the *CMS Pipelines* home page.

: **Significant Documentation Fixes**

: The documentation has been corrected for Version 1 Release 1 Modification Level 12 as
: follows.
:
: *spec* The NUMBER data source was changed in Modification Level 6 sublevel X'85'
: to behave as now documented in this book, but manuals and help files
: remained unchanged.

How to Send Your Comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

To send us your comments, go to *z/VM Reader's Comment Form* and complete the form.

If You Have a Technical Problem

Do not use the feedback method.

- Contact your IBM service representative.
- Contact IBM technical support.
- See IBM: *z/VM Service Resources*
- Go to IBM Support Portal

Summary of Changes

This book is based on *CMS/TSO Pipelines: Author's Edition* SL26-0018, and replaces *z/VM: CMS Pipelines Reference* and *z/VM: CMS Pipelines User's Guide* that apply to previous releases of z/VM.

The revision codes (characters or symbols in the left margin) show changes from previous editions.

A period marks changes for Level 1.1.11.

A colon marks changes for Level 1.1.12.

An exclamation mark marks changes for z/VM 6.4.

The reference sections in this book marked with these revision codes represent new function in *CMS Pipelines* that was not shipped with z/VM 6.3 or earlier z/VM releases.

SC24-6252-01, *CMS Pipelines* 1.1.12/0012

- Support for ECKD devices with more than 65519 cylinders in the *trackread* and *trackwrite* built-in programs.
- A new SECURE option on *tcpclient* and *tcpdata* to write TCP/IP client and server that use z/VM System SSL for secure connections.
- A new built-in program *ftp* to read data from an FTP Server into the pipeline, or write data from the pipeline to an FTP Server.

Part 1. Introduction

This part of the book is for new users of *CMS Pipelines*. No knowledge of pipelines is assumed and only limited CMS or TSO experience is required.

The pipeline concept fits programs together like pearls on a string; as if by magic, output lines written by one program become the input to the next program in the pipeline. *CMS Pipelines* has many built-in programs that may be helpful for the end user when manipulating data.

Thus, the pipeline concept greatly simplifies the task of making commands, tools, and programs in general.

! Chapter 1, “Summary and Two Examples” gives a quick overview and introduces the
! *CMS Pipelines* concept and terminology for users who know CMS or TSO.

Chapter 2, “A Walk Through a Pipeline” shows how to build a small application using *CMS Pipelines*.

Chapter 1. Summary and Two Examples

This chapter is for those who wish to:

- Understand quickly how *CMS Pipelines* fits into CMS and TSO.
- See samples of tasks performed with *CMS Pipelines*.

Summary

In essence, *CMS Pipelines* is a command, PIPE. The PIPE command interprets its argument string as a pipeline specification, which is a list of programs to run. A program has a name and often an argument string. A solid vertical bar (|) marks the end of the specification of one program and the beginning of the specification of the next. Programs are either built into *CMS Pipelines* or written by the user (usually in REXX). There is a *connection* from the *output stream* of the program to the left of the vertical bar to the *input stream* of the program on the right of the vertical bar. The order of programs in a pipeline specification defines how data are passed from one program to the next: data are pumped from left to right in a pipeline.

The pipeline specification is scanned by *CMS Pipelines*, and the programs are started. A particular program can be used several times in a pipeline; each instance of a program in a pipeline is called an *invocation*. An invocation of a program is also called a *stage*. Each stage runs independently of all other ones; there is a *pipeline dispatcher* to coordinate it all and make sure that data flow through the pipeline. Programs obtain data from the pipeline dispatcher or from a *host interface* (an interface to the underlying operating system); they deliver data to the pipeline dispatcher or a host interface. Programs accessing a host interface are called device drivers because the interface often reads or writes a device or file. Programs that do not interact with the host are called *filters*; they process data in the pipeline in some particular way.

Some advantages of a pipeline implementation are:

- Small simple programs are combined to accomplish a task that is often not trivial.
- Programs use a standard interface so that any program can be combined with any other one.
- A program is written without regard to the rest of the world; this means that a pipeline program is simple to write and is more robust than a program written as a stand-alone module.
- A REXX pipeline program can be written without regard to the operating system. A program that does not issue CMS or TSO commands can be moved between the systems without change.

Many Streams

CMS Pipelines supports a network of interconnected pipelines, only limited by the memory size of the virtual machine. Multiple pipelines are specified in a PIPE command. A program using more than one stream is declared with a label on its primary pipeline; subsequent references to the label specify where additional streams are connected to surrounding stages.

Pipelines are added to the set of running ones in two ways. A program can call a subroutine pipeline to process its input data or generate output data, or both; or a program can

transfer a stream to a new pipeline that is added to the set and runs in parallel with the current pipeline set.

Writing Programs

Though you can accomplish many tasks with built-in programs and combinations of built-in programs (cascades of filters), there will no doubt be times when you need a function for which there is no program readily available. You (or someone else) must then write a program to perform this function. Such programs are often written in the REXX language. They can be compiled with the REXX/370 Compiler or run by the REXX Interpreter.

Pipeline programs in REXX read and write the pipeline using pipeline commands; other pipeline commands add pipelines to the set of running pipelines, run a subroutine pipeline, and perform many other functions.

A Pipeline Example

Assume you are giving a presentation and you wish to know the number of words in your manuscript. The manuscript is stored in a text file. Figure 2 shows a way to do this.

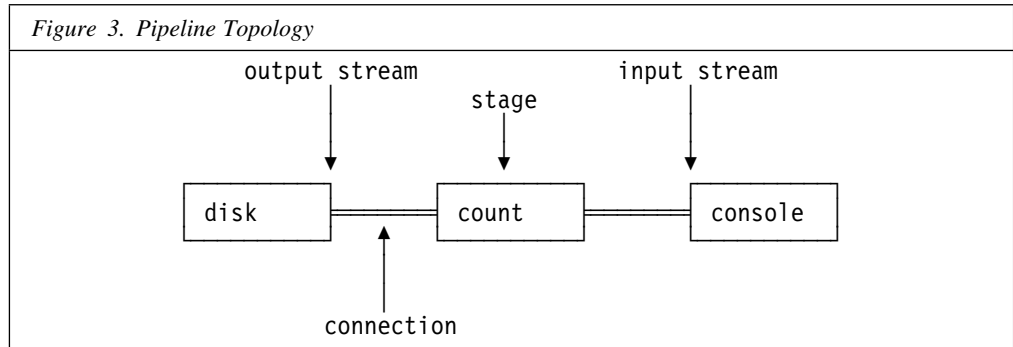
<i>Figure 2. Very First Example: Counting Words</i>
<pre> pipe disk seas88 script count words console ▶3573 ▶Ready; </pre>

The first line shows the PIPE command; the second line is the response from *CMS Pipelines*; the third line is the CMS ready message indicating that the command has completed without error. The arrowhead on the left side of the last two lines is a convention in this book to indicate that these lines are written by the system; the arrowheads are not displayed on your terminal when you issue CMS or TSO commands. The blank in front of the PIPE command indicates that this line is typed on the terminal by the user.

The command specifies that three programs are to be run: one to read a file (*disk*), one to count words (*count*), and the last one (*console*) to display the result on your terminal. The three programs are separated by two solid vertical bars. Because a program in a pipeline is called a stage, and because the bars separate pipeline programs, the solid vertical bar is also called the stage separator.

Figure 3 on page 4 shows a drawing of the pipeline in Figure 2.

Summary



When running the example in Figure 2 on page 3:

1. CMS or TSO looks for the command PIPE and somehow finds the pipeline module.
2. *CMS Pipelines* scans the argument string to see which programs should be selected. It finds three; via an internal table, it resolves the names to built-in programs residing within the pipeline module.
3. The pipeline specification has no errors; thus, the programs are run. Conceptually, the three programs are run in parallel, but in reality, control must pass from one to the other. The pipeline dispatcher takes care of this; you seldom need to concern yourself with the way it is done.
4. Anyhow, it starts the leftmost program first: *disk* is called.
5. *disk* reads the file and calls the pipeline dispatcher to write one line at a time to its primary output stream; the pipeline dispatcher looks for someone to read the line.
6. *count* is started. It calls the pipeline dispatcher to read a record. The two sides of the stage separator are now in a state where a line can be passed from one to the other; the program on the left is writing a record and the program on the right is reading.
7. The pipeline dispatcher passes the line from *disk* to *count*, makes a note that *disk* has now written the line, and runs *count* again.
8. *count* counts the number of words in the line, discards the line, and calls the pipeline dispatcher to read another one.
9. The pipeline dispatcher finds that there is no line being written by *disk*. It suspends (stops running) *count* for a while and resumes *disk* which reads another line from the file and writes it to the pipeline, and so on.
10. Eventually, CMS reflects end-of-file on a call to read from the file. *disk* then returns to the pipeline dispatcher from the call in step 4.
11. *count* is waiting for an input record, but there are no more. The pipeline dispatcher resumes *count* with a return code to indicate end-of-file. *count* now writes a line containing the count of words.
12. The pipeline dispatcher finds *console*, starts it, and passes the line to *console*, which writes the result to the terminal. *console* reads another line from the pipeline, resuming *count*.
13. *count* returns.
14. The pipeline dispatcher reflects end-of-file to *console*, which also stops.
15. All programs are now complete. The pipeline dispatcher returns to CMS (or TSO) with a return code, in this case 0.

Returning to the command in Figure 2 on page 3, note that the first two programs have arguments to indicate what they should do. The first one (*disk*) needs to know the DSNNAME (z/OS) or the name and type of the CMS file to read; it looks for the file on all accessed minidisks and directories. *count* has a keyword option (WORDS) to make it count words; it counts bytes and lines when other options are used.

The response of a number with no accompanying text to explain it may seem a bit terse at first. There are good reasons why *count* writes a number without, for instance, “words” after it; it is simpler to add text than to remove it.

You can tell from the ready message that this particular example was run on CMS. Had it been run on TSO, the command would have read the member SEAS88 from the data set allocated to DDNAME SCRIPT. (There are other ways to read z/OS data sets.) And the ready message would have been all capitals with no trailing semicolon.

The most important observation has been kept to the end. The first example shows the independence of the programs in the pipeline. When first in a pipeline, *disk* reads a file, no matter what is put after it in the pipeline to process the data. Likewise, *console* is not choosy about what it writes to your terminal; there could be anything in between the two programs to process a file and type the result. So what happens if you put nothing between *disk* and *console*? Well, then you have the CMS TYPE command. Looking at it backwards, the sample also shows how easy it is to adapt a pipeline to some other needs:

- Think of a CMS command that almost does what you want.
- Refer to Appendix C, “Implementing CMS Commands as Stages in a Pipeline” on page 918 to see how it is done with a pipeline.
- Add filters to the pipeline to tweak it to perform your task.

Another Example

You are writing a book and have the text stored in several files. To get an indication of its size you wish to count the number of words in the book which you have almost written. You cannot use the command shown in Figure 2 on page 3 directly because the book is made from several files. CMS users store collections of files in a different way than TSO users do; let us treat the two cases separately.

The CMS User

A CMS user might have stored the files on a minidisk (or a directory in the Shared File System) that is currently accessed as mode H. The files have UG somewhere in their file name; the file type is SCRIPT. Figure 4 shows a way to count the words in all of these files.

Figure 4. Another Example (CMS)

```
pipe cms listfile *ug* script h | getfiles | count words | console
▶16111
▶Ready;
```

The approach in Figure 4 is to ask CMS which Script files on disk H have a UG in the name, read the contents of the files into the pipeline, and then count the number of words in the aggregate of the files.

The command is a pipeline that uses the last two programs from Figure 2 on page 3 to count the number of words; the first two stages get the contents of the required files.

Summary

cms has an argument string that looks remarkably like a CMS command. It runs the CMS command to list files and traps the response CMS would normally write to the terminal; in this case it is the list of files in your book.

This list is passed to *getfiles* which reads the contents each of the files into the pipeline. You can visualise this as replacing the name of the file with the contents of the file.

The TSO User

A TSO user might have stored the files in a separate PDS. Figure 5 shows how to count the number of words in all members of a partitioned data set.

Figure 5. Another Example (TSO)

```
pipe listpds dd=ugscr|chop 8|readpds dd=ugscr|count words|console
▶16111
▶READY
```

This pipeline uses *listpds* to read the list of members from the PDS directory of the data set allocated to DDNAME UGSCR, which we assume contains the Script files for the book. The output records contain all information present in the directory; the member name occupies the first eight columns; *chop* truncates the record after the member name.

Thus, the input to *readpds* contains the names of all members of the PDS. *readpds* reads members, one at a time, from the PDS into the pipeline; thus, the output from *readpds* contains all the files in the book. *count* counts the words in the aggregate and *console* displays it on the terminal.

Chapter 2. A Walk Through a Pipeline

This chapter aims to explain *pipethink*: how to solve a problem by dividing it recursively into smaller problems until each can be done with a program that is built into *CMS Pipelines* or a simple program written by the user. Dividing a problem into two problems of equal size often reduces the complexity by more than a factor of four.

We walk you through the development of a small application, explaining things and pointing out important considerations on the way. The commands and programs shown represent what is required to perform the tasks at hand; no attempt is made to give a complete description of all features of each.

TSO users should not be put off by the clear VM/CMS bias of the examples in this chapter. The aim is to show how to craft a pipeline to perform a task and how to adapt it by *stepwise refinement*. The *cp* device driver used in the following example is simply a convenient source of data which are stored in a file to be processed at our leisure. The following examples could have used a report file as their input just as well. The point is that after the *cp* device driver has written the command response into the pipeline, the following stages do not “know” and certainly do not care if their input data come from CP, from a file, or, indeed, from the moon.

The examples you are going to see show how the response to a CP command is processed to provide information not directly available from standard CP and CMS commands. Though few *CMS Pipelines* users are expected to be interested in the application for its own sake, it is hoped that the examples shown illustrate the process of developing a pipeline application.

Getting Data In and Out of the Pipeline

Data must be entered into the pipeline before they can be processed; and once the data are in a fit state, they must be written out of the pipeline. *CMS Pipelines* provides several built-in *device drivers*, programs to interface the pipeline to the outside world: programs to issue commands; programs to read or write files, or both; programs to read from or write to the terminal; and many more. The first step in designing a *CMS Pipelines* application is to select the device drivers to use.

The CP command “query names” displays information about users logged on to your system. CP writes the response to your terminal screen when you issue the command directly to CP, or via CMS which forwards it to CP; the response can also be captured and processed in the pipeline. Figure 6 on page 8 shows how the command is issued with *CMS Pipelines*; the response is written to the file LOGGED USERS A and displayed on your terminal.

Walkthrough

Figure 6. CP Command with Response Logged to Disk and Written on the Terminal

```

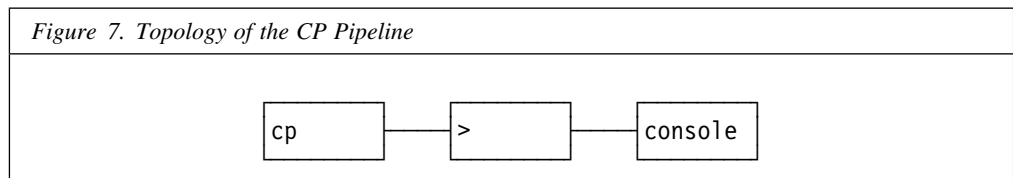
pipe cp Q N | > logged users a | console
▶ICSTAT3 - DSC , FINNS - DSC , EPLDICT - DSC , EPLCS - DSC
▶DKEMERG - DSC , CPO - DSC , CISLTLX - DSC , BUYDBM - DSC
▶ARKIV - DSC , SQLDBA3 - DSC , YVETTE - DSC , VM3ACCT - DSC
▶TRT - DSC , TORSTEN2 - DSC , VANILLA -L0001, IMPOSIT - DSC
▶DRIFTNYT - DSC , VMBCKUP3 - DSC , VMAVAIL - DSC , VMASMON2 - DSC
▶VMASMON - DSC , SQLMON - DSC , SNAOPER - DSC , RSCSPC - DSC
▶RSCS - DSC , PVMB - DSC , PVM - DSC , NETOPER - DSC
▶NET - DSC , ISPVM - DSC , VMOPER - 05A0, COLPRT2 - DSC
▶AP2SVP - DSC , VMAUTO - DSC , VMCLASSI - DSC , SMART - DSC
▶DATAMOVE - DSC , DIRMAINT - DSC , SHRMGR - DSC , AUTPWMON - DSC
▶AUTOOPER - DSC , VMAUDIT - DSC , AUTOLOG1 - DSC , OPERATOR - DSC
▶VMTODDY - DSC , VMBSYSAD - DSC , CARLCH -L0003, JOHLJUNG -L0002
▶TOMMYJ - DSC , EPC -L0000, SCHEEL - DSC , KURTKR - DSC
▶OPRATNSA - DSC , SEN - DSC , POE - DSC , TOMS - DSC
▶SPCPRO - DSC , SPCENT - DSC , SCRSRV - DSC , QASTAT - DSC
▶PESERV - DSC , NPSM - DSC , JOHN - 05A2
▶Ready;

```

What happened? The CMS command PIPE is issued. It scans its arguments and finds two solid vertical bars separating the specifications of three programs to run in a pipeline. Such programs pass data to each other via a standard interface in what is called the pipeline dispatcher in PIPE. A program running in a pipeline is called a *stage*. Of the three stages, the first one, *cp*, issues its argument string as a command to CP and writes the response to what is called the primary output stream, one record for each line in the response. *>* replaces the contents of a file with records read from its primary input stream; on CMS, the arguments specify the file name, type, and mode; on TSO, and with this particular format of the parameter list, the member LOGGED is replaced in the PDS allocated to the DDNAME USERS (possibly not the best choice of names, but it emphasises the portability between CMS and TSO). *>* also copies the file to the primary output stream; *console* reads it and displays it on the terminal.

Figure 7 shows the layout of the pipeline in Figure 6. Each block represents a stage; the line between them represents the connection between output and input streams.

Figure 7. Topology of the CP Pipeline



Recall that each stage communicates with the pipeline dispatcher; a stage does not call a neighbour directly. This simplifies the programming of a pipeline program considerably; for instance, the result of the CP query can be inserted into a DB2 database directly from the pipeline without ever touching a file. Consider that *cp* was written before SQL/DS was even announced; but once the *sql* driver was written for *CMS Pipelines*, all pipeline programs acquired support for SQL/DS (now DB2 Server for VM) without change to a single one of them. Such is the power of the pipeline concept.

Filtering Pipeline Data

The response in Figure 6 on page 8 is typical of a Sunday morning when most virtual machines logged on are service machines that do not represent “live” users. The following examples show how to extract the “interesting” information from the response, but first a general remark. It is a good idea when developing an application to store a reference input file to be used for testing, instead of running test cases based on a command response at the time the test is run. For instance, the number of logged on users is likely to be different on Monday morning. This can expose an error in your pipeline that you have not met before; use a reference for test data to ensure that your tests are repeatable when debugging a pipeline specification.

The first thing one might ask is, who is connected? *CMS Pipelines* has many programs to select lines with or without a string, label, or what not, but the response in Figure 6 on page 8 needs a bit of massaging before it is in a form where the connected users can be selected. Selection stages select complete lines; the response in Figure 6 on page 8 is four abreast and must be split up with a line for each virtual machine.

Use *split* to display each virtual machine on a separate line. *take* selects the first 5 records to limit the number of records shown. Figure 8 shows the command and the response.

Figure 8. Splitting the CP Response

```
pipe < logged users | split , | take 5 | console
▶ICSTAT3 - DSC
▶FINNS - DSC
▶EPLDICT - DSC
▶EPLCS - DSC
▶DKEMERG - DSC
▶Ready;
```

< reads the test reference stored previously and writes it into the pipeline. The comma (,) after *split* makes it split records at commas; the commas are discarded. Thus, lines split off have a leading blank, which explains why the response is ragged. Use *strip* to remove leading blanks as shown in Figure 9; as used here, *strip* also removes trailing blanks, but they are a bit more difficult to see.

Figure 9. Removing Leading and Trailing Blanks

```
pipe < logged users | split , | strip | take 5 | console
▶ICSTAT3 - DSC
▶FINNS - DSC
▶EPLDICT - DSC
▶EPLCS - DSC
▶DKEMERG - DSC
▶Ready;
```

Now you have a line for each virtual machine logged on. The first eight bytes contain the name of the virtual machine; the name is followed by a hyphen (-). The last word is the address of the terminal from which the virtual machine is logged on; DSC is displayed if the virtual machine is disconnected. A *take* stage was added to the pipeline to limit the output for this figure.

Walkthrough

Removing lines where the terminal is shown as DSC excludes disconnected virtual machines from the list; presumably what is left is a list of connected virtual machines. So, Figure 10 on page 10 shows how to see which virtual machines are connected:

Figure 10. Finding Connected Virtual Machines

```
pipe < logged users | split , | strip | nlocate /- DSC/ | console
▶VANILLA -L0001
▶VMOPER  - 05A0
▶CARLCH  -L0003
▶JOHLJUNG -L0002
▶EPC     -L0000
▶JOHN    - 05A2
▶Ready;
```

So far, you have seen how to issue a CP command and process its response, storing it in a file, and displaying it on your terminal. You have seen how to process the response and you have *fine-tuned* a suite of filters to create a command to perform a function not readily available with standard commands.

It is indeed the *CMS Pipelines* way to write long pipelines with many stages, but typing such long commands on the keyboard is not the way to do it. Store pipeline specifications in REXX programs instead. Two types of REXX programs are now presented: subroutine pipelines and “normal” CMS command procedures (EXECs).

Subroutine Pipelines

Today you wish to see who is connected, but maybe some other day you would like to see who is connected at a particular control unit; the sequence of *split*, *strip*, and *nlocate* seems to be something you might do often when processing the response to CP commands.

Figure 12 on page 11 shows how this sequence of commands is stored as a *subroutine pipeline*. On CMS, it is in the file CNCTD REXX; on z/OS, it is the member CNCTD of the PDS that is allocated to the DDNAME FPLREXX. Once the file is created, simply write *cnctd* in the pipeline specification as shown in Figure 11, instead of the three filters you used before.

Figure 11. Using a Subroutine Pipeline

```
pipe < logged users | cnctd | console
▶VANILLA -L0001
▶VMOPER  - 05A0
▶CARLCH  -L0003
▶JOHLJUNG -L0002
▶EPC     -L0000
▶JOHN    - 05A2
▶Ready;
```

The program in Figure 12 on page 11 is a REXX program; it has a comment on the first line to indicate that it is indeed a REXX program. However, it differs from normal command procedures (EXECs) in two respects:

- *CMS Pipelines* resolves the program automatically when the file type is REXX rather than EXEC; *TSO Pipelines* resolves the program from the data set allocated to FPLREXX rather than SYSEXEC. The program is not stored as a normal EXEC because the

commands in the program are pipeline commands, not CMS or TSO commands; having a different file type makes it more difficult to use a program in the wrong context.

- The command itself is probably not like any CMS or TSO command you have seen.

A subroutine pipeline normally contains the single pipeline command CALLPIPE with arguments, followed by the REXX instruction exit to specify the return code from the program as the return code from the subroutine pipeline.

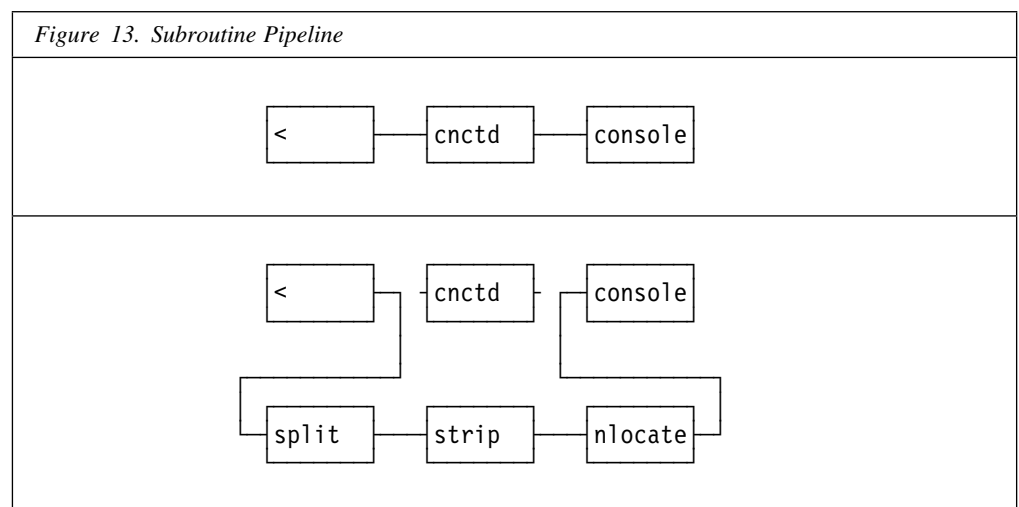
Figure 12. CNCTD REXX: Subroutine Pipeline to Select Connected Virtual Machines

```
/* Select connected virtual machines */
'callpipe *:| split , | strip | nlocate /- DSC/ |*:'
exit RC
```

The command itself probably looks a bit unfamiliar, but you recognise the three programs used to select the connected virtual machines. The asterisk followed by a colon (*:) is called a connection. It is put as a stage at the beginning and the end of the subroutine to indicate that the caller's input is connected to the left side of the subroutine, and that the output of the subroutine is to be written to the caller's output. The command is said to be in *landscape format*² because it is a single line.

Figure 13 shows the layout of the pipeline when it is started and after the subroutine pipeline is active.

Figure 13. Subroutine Pipeline



Use *count* LINES to see how many users are connected. *count* reads all input lines and then writes a single number to its output. Figure 14 shows how.

Figure 14. Counting Connected Users

```
pipe < logged users | cnctd | count lines | console
▶6
▶Ready;
```

² This metaphor is from painting: landscapes are wider than they are tall, while portraits are taller than they are wide.

Walkthrough

Often the address of the terminal is not interesting. Use *chop* to truncate records. Figure 15 on page 12 shows just the names of the users connected. Each line is 8 bytes long with trailing blanks you cannot see.

Figure 15. Listing the IDs of the Users Connected

```
pipe < logged users | cnctd | chop 8 | console
▶VANILLA
▶VMOPER
▶CARLCH
▶JOHLJUNG
▶EPC
▶JOHN
▶Ready;
```

The sequence of *cnctd* and *chop* seems to be so useful that you might wish to write a subroutine to perform this function. You could extend CNCTD REXX to include the *chop* filter and save it under another name, but this is not recommended because you would have two copies of the subroutine to maintain. Subroutine pipelines can be nested to any depth. Figure 17 shows CNCTDN REXX which writes just the names of connected users.

Figure 16. Just the User IDs

```
pipe < logged users | cnctdn | console
▶VANILLA
▶VMOPER
▶CARLCH
▶JOHLJUNG
▶EPC
▶JOHN
▶Ready;
```

Figure 17 shows the subroutine pipeline written in *portrait format* with a line for each stage; there is a comma (,) after each stage to indicate to REXX that the command is continued on the following line. There are comments at the right of the line; REXX can cope with comments after the comma for continuation.

Figure 17. CNCTDN REXX: Subroutine to Write Names Only

```
!
! /* Just the names of the connected virtual machines */
! 'callpipe', /* Pipeline command to call subroutine */
! ' *:', /* Connector for caller's input */
! '| cnctd', /* Call subroutine to find connected only */
! '| chop 8', /* Discard terminal address */
! '| *:', /* Write to caller's output */
! exit RC
```

Two XEDIT macros were used to help write the program in Figure 17. See “Editing Tools” on page 24 for a discussion of FMTP and SCM.

Writing EXECs with Pipeline Commands

Once you have a suite of filters to perform a useful function you may wish to make it a command: simply put the PIPE command in a normal command procedure (EXEC).

You can also use command procedures when developing the pipeline; add stages to a PIPE command in a command procedure as you fine-tune your pipeline specification.

As an example of a command procedure, CNTLOG EXEC counts the number of users logged on and connected to a terminal. It displays the result with a few additional words. The EXEC uses the “live” CP query and the subroutines developed earlier to process the result of a query. Figure 19 shows the EXEC; Figure 18 is an invocation.

Figure 18. Counting Connected Users

```
cntlog
►There are 31 users connected.
►Ready;
```

You recognise the comment on the first line to indicate that the program is written in REXX.

Figure 19. CNTLOG EXEC: Counting Users Logged on

```
! /* Count number of logged-in users */
! signal on novalue
! address command /* Send commands ... */
! 'PIPE', /* ... directly to COMMAND */
! ' cp q n', /* Perform QUERY */
! '| cnctd', /* Find connected users */
! '| count lines', /* Count them */
! '| spec /There are/ 1', /* Literal field */
! ' /*-* nextword', /* The input number */
! '| /users connected./ nextw', /* More literal */
! '| cons' /* Display */
!
! exit RC
```

The first line that is not a comment (`signal on novalue`) tells REXX to branch to the label `novalue` when referencing a variable that has had no value assigned, but there is no such label in the program. This is deliberate to force a syntax error at the point where the variable without a value is referenced; REXX writes the procedures active at the point of failure when a syntax error occurs. The second instruction (`address command`) requests that REXX send commands directly to the CMS command environment; this is where the PIPE command is, so some processing time is saved in the invocation.

If you do not issue the `signal on novalue` instruction, REXX treats a reference to a variable that has not been set as the literal string consisting of the variable’s name in upper case. This can be handy in small simple programs, but it can be the source of subtle errors when a subroutine defines a variable, which causes a different command to be issued somewhere else in the REXX program. Literal constants must be in quotes when `signal on novalue` is active. In Figure 19 the PIPE command is such a literal. Another reason to write a pipeline specification as a quoted string is the abundance of solid vertical bars. They would be interpreted by REXX as inclusive OR operators if they were not put in quoted strings.

Walkthrough

Most stages in the PIPE command in Figure 19 have been described already; *cons* is an abbreviation of *console*. A few program names have an abbreviation, but most must be spelt out.

The text around the number of logged on users is added by *spec*. This is a versatile program that you are going to meet again many times. It tends to have a long argument string; in this case the *spec* stage requires three lines when formatted for comfortable reading. For each input line, *spec* goes through the list of items in its argument string and performs each item once, in the order written. An item specifies a field; it can be data from the input record, or a literal. A literal field is a delimited string; that is, between two occurrences of a delimiter character, which cannot appear in the string itself. An input field is a column range; **-** means the whole input record. The second part of an item indicates where the field is placed in the output record. A number selects a specific column; the keyword NEXTWORD (which can be abbreviated to NEXTW—it has a synonym NWORD that can be abbreviated to NW) appends a blank and the field to the output record. Thus, the *spec* stage used above puts a literal before and after the number of users logged on.

Maybe you would like to see the IDs of the virtual machines. A single line of eight characters for each is a bit unsophisticated; Figure 20 shows how to format the result of the query with eight users per line. *join* joins lines; as used here, 7 lines are appended to a line with a single blank added between each. The blank is written in a delimited string, just like the constants in *spec* in the previous example.

```
Figure 20. Writing 8 Users Across
  pipe cp q n | cnctdn | join 7 / / | console
▶VMOPER MET RONNING BRANDSEN WILKEN FINNPED BJS HOBERG
▶CFH SOEGAARD BARNER1 SORENSV SVENSSON LAURSEN OTTOH CDJ
▶LESLEY BRINK TOMHRAS HENJOR FFMAINT ABHOUG BENTEP RUDY
▶LOGL0005 THUESEN SCHWANEN FRANZW MHVIID BARNER WOEBBE SPCTOOL
▶JOHN
▶Ready;
```

But what's that? Who is LOGL0005? Quickly store another test case:

```
Figure 21. Saving a New Test Reference
  pipe cp q n | cnctd | > many users a | find LOG | console
▶LOGL0005 -L0005
▶Ready;

  pipe < many users | count lines | console
▶32
▶Ready;
```

The test case is stored; *find* selects the offending line to make sure it is in the new test case. The count of lines shows that someone logged off before you could capture the new test case, but it seems to have the data you need.

The line means that terminal L0005 is in the state where no one is logged on, but the VM logo is not displayed. In this state, a user can send and receive messages without being logged on, for instance to ask the operator to call on the phone. So, as far as CP is concerned, this represents a user even though the true identity is not known. Most of the

time, however, no one sits at the terminal; it has been left in this state after the previous session. You do not wish to include such a terminal in the list of logged on users.

How can you exclude lines for virtual machines that have no user logged in? If your system has no users whose IDs begin with “LOGL”, you can use *nfind* to exclude lines beginning with this string as shown in Figure 22. (Because the argument to *nfind* has three trailing blanks, only user IDs that are seven or eight characters are discarded.)

Figure 22. Discarding &u\$LOGLxxx Machines.

```
pipe < many users | nfind LOGL   | count lines | console
▶32
▶Ready;
```

! But if one of the users of your system is Logland, then this approach is too simplistic.
! The user IDs to exclude are of the form LOGLxxx where xxx is the same as the device
! address. So, which *CMS Pipelines* filter excludes lines where the right characters of the
! user ID are the same as the terminal address? Though *pick* can do this, let us assume there
! is no such filter; you must write one yourself.

Writing a REXX Program to Process Data in the Pipeline

All is not lost when the programs provided with *CMS Pipelines* do not provide a function you need: you can write filters for the pipeline in REXX. Figure 23 shows how to invoke *realuser* to discard lines for terminals in the state between logo and logged on.

Figure 23. Counting Real Users

```
pipe < many users | realuser | count lines | console
▶31
▶Ready;
```

! Figure 24 shows *realuser*, a REXX program to exclude lines for users of the class described
! above.

Figure 24. REALUSER REXX: Selecting Real Users

```
/* Select only real users. */
signal on novalue
do forever
  'readto in'          /* Read a record into variable "in" */
  If RC ^= 0          /* Are we at EOF? */
    Then exit         /* Yes, quit. All is done */

  parse var in userleft +4 userright +4 '-' +2 addr .

  if userleft^=='LOGL' | userright^==addr      /* Want it? */
    then 'output' in                          /* Yes, keep it */
end
```

The REXX program REALUSER REXX has the same file type as a subroutine pipeline; the two are the same as far as REXX and *CMS Pipelines* are concerned, though they may look different to you.

Walkthrough

This program iterates reading an input line into a REXX variable, testing it, and writing the input line to the output if it does not look like a user ID of the type you wish to exclude.

The instruction `do forever` opens the iteration; it is closed with the `end` instruction. Inside the loop, the variable `in` receives the contents of the next record in the pipeline each time the command `READTO IN` is issued. Note that the name of the variable to receive the value is a literal. It is important to write the name in a way where its value is not substituted by REXX. `READTO` sets the variable as a side effect. REXX sets the variable `RC` to the return code. End-of-file is indicated by a return code 12.

The `Parse` instruction separates the components of the user ID. The leftmost four characters of the user ID go into one variable, the rightmost four go into another variable. The hyphen in quotes instructs REXX to skip to the position after the next hyphen (or the end of the variable). Finally, the rightmost part of the address is assigned to a variable. `Parse` has done the hard work; it only remains to test if the leftmost four characters are not equal to the constant "LOGL" or if the rightmost four characters are not equal to the device address.

The input record is written to the output using the command `OUTPUT` unless the test fails. Note the difference between the `READTO` and the `OUTPUT` command. `OUTPUT` writes its argument string to the pipeline. You can compute an expression to write to the pipeline; for instance, putting a timestamp in front of each record. You can also write a constant. In this example, the input record is copied unchanged to the output if it is selected.

Having written the function to suppress these unwanted lines, add the filter to the subroutine pipeline in `CNCTD REXX`. Figure 25 shows the subroutine converted to portrait form. Note that the new function is retrofitted to all uses of `cnctd`. Closer inspection of the output also shows entries with "VSM" in our list. This is not an actual user on the system, but rather a network service machine entry. We add a `nfind` stage to keep that out of our list as well.

Figure 25. Retrofitting `REALUSER` to `CNCTD REXX`

```
/* Select connected virtual machines */
'callpipe',
  '| *:', /* Read from input stream */
  '| split ,', /* Split to one user per line */
  '| strip ', /* Remove leading and trailing blanks */
  '| nlocate /- DSC/', /* Not the ones disconnected */
  '| nfind VSM ', /* Not the network service */
  '| realuser', /* And not the ones with LOGO cleared */
  '| *:'
exit RC
```

Returning to the list of logged on users in Figure 20 on page 14, do you wish it sorted ascending instead? Figure 26 on page 17 shows the new test case sorted ascending. `cnctdn` trims the terminal address from the line; it does not matter that the lines are already split with one per user.

Figure 26. List of Users, Sorted by ID

```

pipe < many users | cnctdn | sort | join 5 / / | console
▶ABHOUG  BAP      BOBAY  BROCKS  CARSTENG CSS
▶DITHMAR  FINNH   GBJ    HENRIKO HEY      JMLO
▶JOHN     KJELD   KUMMEL LAURSEN MIE      MIH
▶OBR      PEETZ   PERLOK PETERHJ POE      PSIE
▶RENEH    RUDY    SCHWANEN VMOPER  VNETFIX WHC
▶WILKEN
▶Ready;

```

It is undeniably sorted, but you want the presentation to be transposed so that user IDs are ordered ascending in the columns as you see it in Figure 27.

Figure 27. Transposing a Multicolumn Display

```

pipe < many users | cnctdn | sort | pad 9 | snake 6 | console
▶ABHOUG  DITHMAR  JOHN    OBR      RENEH    WILKEN
▶BAP      FINNH    KJELD   PEETZ    RUDY
▶BOBAY    GBJ      KUMMEL  PERLOK   SCHWANEN
▶BROCKS   HENRIKO  LAURSEN PETERHJ  VMOPER
▶CARSTENG HEY      MIE     POE      VNETFIX
▶CSS      JMLO     MIH     PSIE     WHC
▶Ready;

```

pad ensures that each record is 9 characters. *snake* formats a “page” to put the input data into six columns. The column depth is adjusted to ensure that all columns contain at least one line of data.

Issuing CMS Commands

A list of user IDs is not what you want; you wish to see the names of the users logged on. Assume that COTTAGE NAMES is a “names” file for your system. Figure 28 shows who are logged on.

Figure 28. Who Is Working in the Cottage?

```

pipe cp q n | > cottage logons a | console
▶SMILEY  - 0A4, WISTFUL - DSC, DUMMY   - 0A1, GROUCHY  - 0A7
▶OPERATOR - DSC, BOSS   - 0A3
▶Ready;

```

Figure 29 on page 18 shows a display of the names of users logged on, based on a names file.

Walkthrough

Figure 29. Convert a User ID to a Name

```
pipe < cottage logons | cnctd | user2nam | console
▶H. A. Haas
▶S. A. What
▶E. B. Scrooge
▶T.O.P. Banana
▶Ready;
```

`user2nam` in Figure 30 shows how it was done. The program issues the CMS command to look a user up in a names file, writing the first line of the response to the pipeline. It writes a line of question marks if the NAMEFIND command should produce no output. Thus, `user2nam` writes one output record for each input record.

Figure 30. USER2NAM REXX: Substitute Name of User for Id

```
/* Convert userids to names                                     */
signal on novalue
signal on error
do forever
  'readto in'
  parse var in userid . +8
  'callpipe',
    ' cms namefind :userid' userid ':name ( file cottage',
    ' append literal ???',
    ' take 1',
    ' *:'
end
error: exit RC*(RC-12)
```

The command NAMEFIND is issued for each input line. It instructs CMS to look in COTTAGE NAMES for an entry describing the user ID, and to display the name of the user. The response is trapped by *CMS Pipelines* and written to the output from *cms*. (The somewhat cryptic *append literal* ensures that a default is provided in case NAMEFIND produces no response; *take* ensures the response is precisely one line.) Names files are described further in *z/VM: CMS Primer*.

Multistream Pipelines

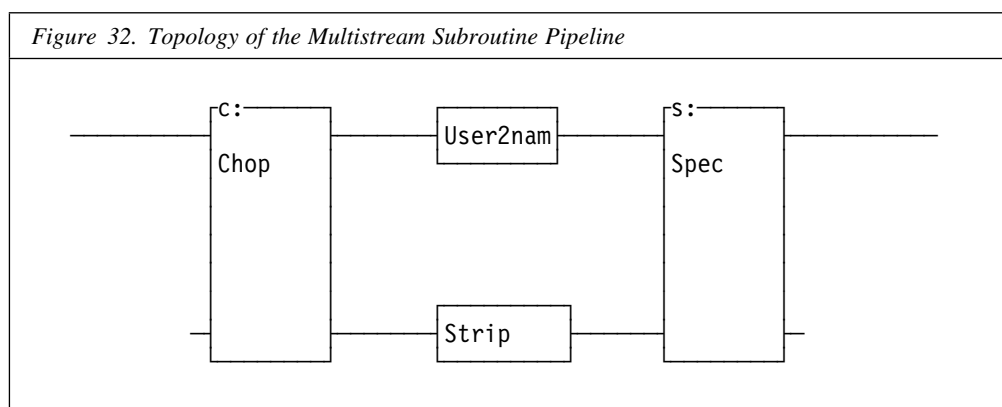
But you really want a display with a line for each connected user. The line should have the terminal address followed by the user name. There does not seem to be any way to make NAMEFIND do this, so more craftiness is required; Figure 31 shows the result.

Figure 31. Display Terminal Address with Name

```
pipe < cottage logons | cnctd | userterm | console
▶0A4      H. A. Haas
▶0A1      S. A. What
▶0A7      E. B. Scrooge
▶0A3      T.O.P. Banana
▶Ready;
```

Figure 33 on page 19 shows the multistream subroutine pipeline to make this display. The approach is to split the line in two parts, the user ID and the terminal address. Each part is processed by itself: the name as you have already seen; the address of the terminal

is shifted to the left. The lines are merged at the end of the subroutine pipeline. Figure 32 on page 19 shows the topology of the pipelines. The primary pipeline is at the top and the secondary is at the bottom. *chop* and *spec* are shown as tall stages because they can transmit data on both pipelines. *chop* only reads from the primary input stream; the secondary input stream is left unconnected. In the same way, *spec* reads from all input streams but writes only to the primary output stream. The *spec* stage shifts the name right to column 10 and inserts the terminal address in the first columns. *strip* removes leading blanks and hyphens from the terminal address (or LU) to make it begin in column 1 when it arrives at the secondary input stream to *spec*.



Pipeline commands are strings, and as such inherently linear; how is the multistream topology transformed to an argument string?

There are two pipelines in Figure 32. The topmost is defined first; the bottom pipeline is defined after the *end character*. The program USERTERM REXX is shown in Figure 33.

Figure 33. USERTERM REXX: Display User Name with Terminal Address

```

!
! /* Display user name with terminal address. */
!
! signal on novalue
! 'callpipe (end ?)',
!   | *: ', /* Input records here */
!   | c:chop 8', /* Split after column 8 */
!   | user2nam', /* Get the name of the user */
!   | s:spec 1-* 10', /* Shift response right */
!     'select 1', /* Read other stream */
!     '1-* 1.8', /* Put first */
!   | *: ', /* To the output */
!   '?c:', /* The rest of the input record */
!   | strip leading anyof /- /', /* Remove - and blanks */
!   | s:' /* Pass to spec */
!
! exit RC

```

Parentheses at the beginning of the argument string to CALLPIPE contain a *global option* to control how CMS Pipelines scans the argument string. This one defines the question mark (?) to be a special character to delimit pipelines, called the end character.

chop was used in Figure 15 on page 12 to truncate records after eight characters. It writes the truncated record (the part of it to the left) to the primary output stream; if the

Walkthrough

secondary output stream is defined and connected, the part of the record chopped off is written there. In this example, the first position of the record on the secondary output stream is a blank or a hyphen, depending on how the terminal is attached. *strip* removes leading blanks, hyphens, or both. The delimited string after the keyword *ANYOF* specifies a list of characters to be removed from the beginning of the record. *strip* repeatedly removes a leading character that is present in the enumerated list of characters; it stops at the first character that is not in the list.

spec takes the name from the primary input stream and combines it with the device address on the secondary input stream. The resulting line is written to the primary output stream.

A DB2 Query

CMS Pipelines interfaces to DB2 Server for VM if your virtual machine is registered with DB2 and you have run the *SQLINIT* procedure. *TSO Pipelines* interfaces to DB2 for z/OS in a similar way. Records are passed between DB2 and *CMS Pipelines* without changing the format of fields (for instance, integers are not made printable in a query). This gives you complete control over the format of data going to and from the database, but on the other hand you must worry about data formats and null values in general. Conversely, *sqlselect* formats a query with column headings and converts data from internal representation to character strings. Figure 34 shows a query in one of the sample databases.

Figure 34. SQL Query

```
pipe sqlselect * from sqldba.inventory where description='BOLT' | cons
▶PARTNO----- DESCRIPTION----- QONHAND-----
▶      221 BOLT                      650
▶      222 BOLT                      1250
▶Ready;
```

For further information, refer to Chapter 11, “Accessing and Maintaining Relational Databases (DB2 Tables)” on page 138 and to *sql*.

Part 2. Task Oriented Guide

This part of the book explains how to use *CMS Pipelines*, showing many examples using built-in programs.

Chapter 3, “Where Do I Start?” on page 22 explains how you can verify that *CMS Pipelines* is installed; it gives hints on how to obtain help; and it describes some tools that may be useful when creating command procedures that use *CMS Pipelines*.

Chapter 4, “Building a PIPE Command” on page 29 explains how to connect the pipeline to your terminal and to your files; it explains how to issue commands and process their responses in the pipeline; and it shows many examples of built-in programs that filter data.

Chapter 5, “Using Multistream Pipelines” on page 74 explains how to specify a network of interconnected pipelines with programs that support more than a single input and output stream.

: Chapter 6, “Processing Structured Data” on page 91 describes how to refer to data in
: records symbolically rather than by column, word, or field number.

Chapter 7, “Writing a REXX Program to Run in a Pipeline” on page 97 explains how to write a REXX program to process data (a REXX filter).

Chapter 8, “Using Pipeline Options” on page 120 explains how to specify options that control the pipeline itself.

Chapter 9, “Debugging” on page 124 explains how to cope with trouble in the pipeline (the data plumber’s guide to blocked drains).

Chapter 10, “Pipeline Idioms—or—Frequently Asked Questions” on page 127 explains some pipeline idioms and annotated answers to some frequently asked questions.

Chapter 3. Where Do I Start?

In this chapter you see how to make sure that *CMS Pipelines* is installed and available to you and how to use your terminal with *CMS Pipelines*. We also show some tools you may find useful on CMS.

IBM Manuals

The present book covers everything you need to know to use *CMS Pipelines* from your terminal and when you write REXX programs to issue pipeline commands or process data.

“Bibliography” on page 957 lists other books you may find useful.

Tutorials and Papers

There are several excellent papers on the *CMS Pipelines* home page, in particular *Plunging into Pipes* and *Plunging On*.

<http://vm.marist.edu/~pipeline>

Refer to “Additional Information, Download Site” on page xx for additional pointers.

Ensure *CMS Pipelines* Is Installed

Trying things on your terminal is the best way to learn to use *CMS Pipelines*. Issue the command “pipe query” to see if *CMS Pipelines* is available. Figure 35 shows the response when you are set to go. The first example shows the response of *CMS Pipelines* on z/VM 6.4. The second example shows the response of *TSO Pipelines*; you see this response (with a suitably modified ready message) when using the “runtime library” on CMS.

Figure 35. Determine If *CMS Pipelines* Is Installed

```
pipe query
▶FPLINX086I CMS Pipelines, 5741-A07 1.0112 (Version.Release/Mod) -
▶Generated 20 Jan 2016 at 08:53:20
▶Ready;

pipe query
▶CMS/TSO Pipelines, 5654-030/5655-A17 1.0112 (Version.Release/Mod) -
▶Generated 12 Jan 2010 at 12:50:28.
▶READY
```

This world is not perfect, however; you could see other responses (though that would be highly unlikely on CMS). Figure 36 shows what may go wrong. Contact your system support staff to install *CMS Pipelines*.

Figure 36. Response When *CMS Pipelines* Is not Installed

```
pipe query
▶Unknown CP/CMS command

pipe query
▶COMMAND PIPE NOT FOUND
▶READY
```

Find the Stage Separator on Your Terminal

The default symbol used for the stage separator is the solid vertical bar (|). It appears as such on 3270 terminals attached to a U.S. control unit or a control unit with a UK language diskette.

There are too many variations to list what the solid vertical bar is on other terminals and PC terminal emulators; in all cases, it is the character used as the OR operator in a REXX expression. The solid vertical bar has the code point X'4F', which is displayed as an exclamation mark (!) on many European and Latin American terminals. Some PC keyboards do not have a solid vertical bar. Instead, the terminal emulator maps the split vertical bar (|) into the solid vertical bar. Create an EXEC like SAYBAR (see Figure 37) if you are in doubt what the solid vertical bar is on your terminal.

Figure 37. SAYBAR EXEC Displays the Stage Separator on Your Terminal

```
/* SAYBAR EXEC */
say 'The solid vertical bar is:' '4f'x'.

saybar
▶The solid vertical bar is: |.
▶Ready;
```

Note: 3270 terminals in some countries have both a solid vertical bar (|) and a split vertical bar (|). The solid vertical bar is the stage separator on such terminals.

TSO Logon Procedure

TSO Pipelines requires several data sets to be allocated. You can allocate them in the logon procedure or in some other procedure.

FPLREXX REXX filters are resolved from the DDNAME FPLREXX.

FPLHELP Help information is stored in the library allocated to DDNAME FPLHELP.

: SYSTSPRT REXX issues error messages from reentrant environments to this DDNAME. The
: importance of allocating this data set cannot be overstressed.

STEPLIB The library that contains the PIPE load module, if it is not in Link Pack Area or in the link list.

You should also issue PROFILE WTPMSG. When you have, REXX is at least able to remind you if you forget to allocate SYSTSPRT.

Pipe Help

Help files are included with *CMS Pipelines*; “pipe help menu” displays the help menu for built-in programs.

There is help for each of the programs listed in the inventory. As an example, “pipe help <” displays help for the device driver to read a file.

CMS Pipelines stores information between commands. This includes a list of the last eleven messages issued and the last eleven SQL error codes received. Help for messages is most conveniently obtained through the pipeline infrastructure: the command “pipe help” invokes help for the last message issued. “pipe help 1” invokes help for the second to last message issued, and so on. There are 11 messages in this memory. Help for message

Editing Pipelines

11 is displayed if you type “pipe help 11”; the number is taken to be a message number when it is larger than 10. *TSO Pipelines* is unable to display help for SQL as z/OS does not provide the table of messages.

Issue “pipe help msg <number>” to get help for the message with the number specified. Often the return code is the same as the last message issued.

Type the commands shown in Figure 38 on your terminal to try out the help facilities.

Figure 38. Try Some Help Commands

```
pipe help menu
pipe help disk
pipe zz
pipe help
pipe help 17
```

The CMS HELP command can also be used to display information about *CMS Pipelines* built-in programs and messages, just like for other CMS commands.

Editing Tools

CMS Pipelines users soon find themselves entering pipeline specifications in EXECs or CLISTS. The pipeline specifications become longer and more complex as a “plumber” gains experience.

To help with this task, *CMS Pipelines* supplies two edit macros that you may find improve your productivity when editing pipeline specifications. These macros support the editors ISPF on TSO and XEDIT on CMS.

FMTP converts a pipeline from *landscape* format where the pipeline is specified on a single line to *portrait* format where each stage is stored in a separate record. Having one line per stage means that you can easily add, delete, or move stages in a pipeline specification. It also means that there will be room to the right of the line for a running commentary.

SCM lines comments up nicely on the right. It also adds the ending **/* when a comment is not terminated.

On CMS, the macros are intended to be used from the prefix area.

Using FMTP

FMTP XEDIT is a useful tool when you are entering a pipeline specification into an EXEC. Simply insert the PIPE followed by an option string and some stages:

Figure 39. A Landscape Pipeline

```
TEST1 EXEC A1 V 132 Trunc=132 Size=2 Line=0 Col=1 Alt=3
====>
===== /* My very first test program */
===== 'pipe (end ?) literal Hello, World! | xlate upper | console'
```

Now move the cursor to the prefix area of the line containing the command and type *fmtp*:

Figure 40. Preparing to Convert to Portrait

```
TEST1  EXEC    A1 V 132 Trunc=132 Size=2 Line=0 Col=1 Alt=3
====>
===== /* My very first test program */
fmtp= 'pipe (end ?) literal Hello, World! | xlate upper | console'
```

Then press ENTER to run the `FMT`P `XEDIT` macro.

Figure 41. Portrait Pipeline

```
! TEST1  EXEC    A1 V 132 Trunc=132 Size=5 Line=6 Col=1 Alt=6
!
! =====>
! ===== /* My very first test program */
! ===== 'PIPE (end ? name TEST1.EXEC:2)',
! ===== '?literal Hello, World! ',
! ===== '| xlate upper ',
! ===== '| console'
```

You can see that the macro has converted the pipeline specification. The commas at the end of the lines indicate continuation to `REXX`; though the pipeline specification is now spanned over four lines, it is still just a single `REXX` expression. `FMT`P also added the option `NAME` to identify the line of the program containing the pipeline specification; you will find this very useful when debugging an oil refinery of pipes because error messages will refer you to the file containing the pipeline specification issuing an error message.

The number in the option `NAME` is the line number where the pipeline specification started when the `FMT`P was used to format the code. When you continue to add stages to the pipeline specifications in the program, the number stated in the option `NAME` will often not match the actual line number anymore. This also happens when you maintain your program with `EXECUPDT` and block comments are excluded from the executable form. There is no need to correct the numbers each time you make changes as long as you remember to search for the reference rather than expect it to be exactly at that line in the program.

This particular style having the stage separators to the left is sometimes called a left-handed pipeline. It is the favourite style of most *CMS Pipelines* users because the aligned stage separators make it easy to see the structure. Earlier versions of `FMT`P put the bars to the right of the line. To `REXX` it makes little difference; it is still all just a character string.

Add lines to insert stages in the pipeline. You can insert a landscape pipeline segment and then convert it to portrait form. Remember to begin with a quote and end with a comma to indicate continuation:

Figure 42. Adding a Landscape Segment to a Portrait

```
! TEST1  EXEC    A1 V 132 Trunc=132 Size=5 Line=6 Col=1 Alt=12
!
! =====>
! ===== /* My very first test program */
! ===== 'PIPE (end ? name TEST1.EXEC:2)',
! ===== '| literal Hello, World! ',
! ===== '| xlate upper ',
! ===== fmtp= '| reverse | xlate lower | count words',
! ===== '| console'
```

Editing Pipelines

Press ENTER.

Figure 43. Portrait Pipeline with Added Segment

```
! TEST1 EXEC A1 V 132 Trunc=132 Size=5 Line=6 Col=1 Alt=15
! =====>
! /* My very first test program */
! ===== 'PIPE (end ? name TEST1.EXEC:2)',
!          '| literal Hello, World!',
!          '| xlate upper',
!          '| reverse',
!          '| xlate lower',
!          '| count words',
!          '| console'
```

Isn't all that blank space to the right inviting? Add comments!

Using SCM

The macro SCM XEDIT shifts REXX comments to align them on the right. Simply type the beginning of the comments wherever convenient:

Figure 44. Portrait Pipeline

```
! TEST1 EXEC A1 V 132 Trunc=132 Size=5 Line=6 Col=1 Alt=22
! =====>
! /* My very first test program */
! ===== 'PIPE (end ? name TEST1.EXEC:2)',/* The pipeline command
!          '| literal Hello, World! ',/* Get some data
!          '| xlate upper ',/* Make it uppercase
!          '| reverse ',/* Turn it round
!          '| xlate lower ',/* And force it low
!          '| count words ',/* Have we still two words?
!          '| console'/* Let's see
```

Then position the cursor on the prefix area of the line containing the PIPE command and enter the prefix command to format the lines:

Figure 45. Portrait Pipeline

```
! TEST1 EXEC A1 V 132 Trunc=132 Size=5 Line=6 Col=1 Alt=22
! =====>
! /* My very first test program */
! scm9= 'PIPE (end ? name TEST1.EXEC:2)',/* The pipeline command
!          '| literal Hello, World! ',/* Get some data
!          '| xlate upper ',/* Make it uppercase
!          '| reverse ',/* Turn it round
!          '| xlate lower ',/* And force it low
!          '| count words ',/* Have we still two words?
!          '| console'/* Let's see
```

Press ENTER.

Figure 46. Portrait Pipeline

!
!
!
!
!
!
!
!
!
!

```

TEST1  EXEC  A1 V 132 Trunc=132 Size=5 Line=6 Col=1 Alt=22
====>
===== /* My very first test program */
===== 'PIPE (end ? name TEST1.EXEC:2)', /* The pipeline command */
===== '| literal Hello, World! ', /* Get some data */
===== '| xlate upper ', /* Make it uppercase */
===== '| reverse ', /* Turn it round */
===== '| xlate lower ', /* And force it low */
===== '| count words ', /* Have we still two words? */
===== '| console' /* Let's see */

```

Voilà!

Issuing the PIPE Command from a FILELIST Panel

The PD EXEC shown in Figure 47 saves some keystrokes when you wish to issue a PIPE command against a file the name of which is displayed on a line of a FILELIST panel.

Figure 47. PD EXEC

```

/* Pipe disk read of current file */
parse arg file '|'+0 pipe
address command
'PIPE (end \)',
'<' file '|unpack' pipe
exit RC

```

Specify the file name followed by the remainder of the pipeline specification:

Figure 48. Issuing the PD Command

JOHN	FILELIST	A0	V	108	Trunc=108	Size=196	Line=1	Col=1	Alt=1
Cmd	Filename	Filetype	Fm	Format	Lrecl	Records	Blocks	Date	
pd /	 count words	 console			103	3940	43	11/07/92	1
GETSCR	REXX	A1	V		71	14	1	11/07/92	1
FROM	FULLSCR	A1	V		2604	1	1	11/07/92	1
LOAD	MAP	A5	F		100	799	20	11/07/92	1

Sample Pipelines and REXX Filters

:

Refer to the *CMS Pipelines* homepage for the current samples.

Many of the samples are also shipped with z/VM on MAINT's 193 minidisk.

Compatibility Between TSO Pipelines and CMS Pipelines

There are some differences between the facilities available in the two implementations and some facilities are not available in both implementations. Particular restrictions or lack of support are noted where applicable in the reference part of this book.

These general considerations apply:

Compatibility

- A Virtual Machine on VM/CMS is equivalent to an Address Space on z/OS. For example, SQL runs in a separate virtual machine; DB2 runs in a separate address space.
- Device drivers are available in both environments, where the function can be provided. A CMS file is equivalent to a z/OS physical sequential data set or a member of a partitioned data set.
- Filters, gateways, and all other built-in programs are independent of the operating environment; they are thus generally available.
- REXX filters that issue only pipeline commands (that is, do not use the Address instruction) are directly movable between TSO and CMS without modification. For instance, *sqlselect*, which is implemented as a REXX filter, runs in both environments without code specific to either environment.

See Appendix F, “Pipeline Compatibility and Portability between CMS and TSO” on page 933 for more information.

Chapter 4. Building a PIPE Command

This chapter contains a task oriented selection guide to some of the programs built into *CMS Pipelines*. Once you know which stage you need, look up the syntax definition in the inventory or in the online help. This chapter describes some device drivers and filters you may consider to perform a given task.

Although you can type a complete pipeline specification as a command at your terminal, it is often easier to write an EXEC to do a given function. Such an EXEC includes a pipeline specification usually issued by the PIPE command. On CMS, the PIPE command should be issued to the COMMAND environment (Address `command`); use the ATTACH environment (Address `Attach`) or the LINK environment (Address `link`) on TSO. The EXEC can issue additional CP and CMS (or TSO) commands to complement the pipeline function performed.

The return code from PIPE is the “worst” of the return codes received from each of the stages. If any stage’s return code is negative, then the PIPE return code is the minimum of all stages’ return codes; otherwise it is the maximum one.

Using Device Drivers to Get Data in and out of a Pipeline

The device drivers you should look at first have the same name as the device: *disk*, *console*, *punch*, *reader*, and *tape*. Printer output is done with *printmc*; this reminds you that carriage control is needed. Other device drivers read or write more exotic devices, and some destinations are not devices at all: the stack, XEDIT, and (sub)command environments.

Reading and Writing CMS Files

Use *disk* to read or append to a CMS file on a minidisk or in the Shared File System.

The file is read when *disk* is first in the pipeline specification; otherwise the file is appended. For reading, file name and type is the minimal specification. This suffices when you wish to append to an existing file. (Use `>` if you wish to replace an existing file; *disk* does not erase a file.) You may write a record format after the file mode. For fixed format files you can also specify the record length. For a new file, the default is variable format; it is ensured that an existing file is compatible with the format and record length you specify. The following example shows one way to copy a file to another minidisk (making the new file variable record format). The syntax is good for FILELIST : the output disk mode is the first argument; it is followed by the name, type, and mode of the input file.

Figure 49. Copying a Minidisk File

```
/* copy a CMS disk file */
arg fmo fn ft fm .
if left(fmo,1)=left(fm,1)
    then exit 0          /* copy to self?          */
'pipe <' fn ft fm '|' >' fn ft fmo
exit RC
```

You can inadvertently append to a file by putting a stage in front of the one intended to read the file; the combination of *literal* and *disk* is particularly alluring. To guard against this, `<` is an entry point to *disk* that issues an error message if it is not first in the pipeline,

Using Device Drivers

thus ensuring that the file is always read. Conversely, > and >> must not be first in a pipeline because the two programs replace a file and append to a file, respectively. When using >, you must specify the file mode of the disk to receive the file. Note that a blank character must delimit the command verb from the file name.

To update a file on CMS, it is possible to use < and > for the same file in a pipeline specification. When the file exists, > writes a utility file and does not erase the existing file before processing is complete. Figure 50 shows an example.

Figure 50. Updating a Minidisk File

```
pipe literal a line | > a file a
▶Ready;

pipe < a file a | literal another line | > a file a
▶Ready;

pipe < a file a | console
▶another line
▶a line
▶Ready;
```

This processing is safe as long as the input file is completely read before > receives end-of-file on its input. Using the secondary output stream from, for instance, *take* or *drop* can cause the output file to be created too early. In such a case you can use *buffer* to ensure that the file is read completely before being processed. When the file is too large to buffer in storage, you must write a REXX program that creates a utility file and renames it after the pipeline has completed. You can use the pipeline command COMMIT 1 in a subroutine pipeline to test if all data transport has completed without error before you erase the original file.

When replacing a large file, consider erasing the existing file before starting the pipeline if the existing file is not needed to create the new file. This reduces the disk space required because two copies do not exist when the new file is created. On the other hand, this has potentially undesirable consequences for SFS files that are accessed through a mode letter; refer to the usage notes for >.

You can read and write a file that is stored in the Shared File System (SFS) in two ways. You can use the ACCESS command to access the directory as a mode letter and then refer to the file using the mode letter or you can use the directory path directly without accessing the directory first.

When you use a mode letter, *CMS Pipelines* uses the original minidisk interface to the file system, even when the file is in SFS.

When you specify a directory path, *CMS Pipelines* uses the callable interface to SFS.

Reading and Writing MVS Files

On z/OS these device drivers are used to access physical sequential data sets and members of partitioned data sets:

- < Read a file or a member.
- > Rewrite a file or a member.
- >> Append to a file. z/OS does not support appending to an existing member of a partitioned data set.

disk is also available on TSO; it behaves as < when it is first in a pipeline and as >> when it is in other positions.

A data set can be specified in several ways:

By data set name: When the data set name is not enclosed in single quotes, *TSO Pipelines* applies the prefix, if any has been set by the TSO command SET PREFIX. A member name can be specified in parentheses after the data set name. These are ways to read from a data set:

Figure 51. Reading a z/OS Data Set

```
pipe < names.text | ...
pipe < 'sys1.maclib(time)' | ...
```

To replace a physical sequential data set or a member of a partitioned data set:

Figure 52. Replacing a z/OS Data Set

```
pipe ... | > names.text
pipe ... | > tso.log(test1)
```

The data set must be cataloged when it is referenced by name.

By DDNAME: An already allocated data set can be referenced by its DDNAME. The DDNAME is prefixed by the keyword DDNAME= or any abbreviation down to DD=. A member can be specified in parentheses after the DDNAME. This usage is parallel to the way members are specified with DSNAMES.

Figure 53. Referencing a Data Set by DDNAME

```
pipe < ddname=rexx(tester)
pipe ... | > ddname=sysut1
pipe < c admsymb1 | ...
```

The last line shows the “CMS-compatible” way to specify a member of a PDS that is already allocated to a DDNAME. It reads the member C from the data set allocated to ADMSYMBL. This follows the GDDM standard for the symbol set library.

OpenExtensions Text Files

If OpenExtensions (called USS on z/OS) is available on your system, *CMS Pipelines* will support it in several ways, in particular to read, append, and replace text files. A *text file* contains lines that are terminated with the X'15' newline character.

To access such files, use the <, >>, and > device drivers as if there were nothing special about the file. *CMS Pipelines* inserts line end characters when you write a file and it deblocks the file automatically when you read it.

Using Device Drivers

When reading or writing a hierarchical file, the argument is a single word or a quoted string, which specifies the path to the file using the normal OpenExtensions conventions. A path that begins with a forward slash (/) specifies the fully qualified path from the root of the file system; a path that omits the leading forward slash is relative to the present working directory.

To read the file `sample.c` from your current working directory on CMS and using the full path:

Figure 54. Reading OpenExtensions Files on CMS

```
pipe < sample.c | count characters lines | console
pipe < /u/john/sample.c | count characters lines | console
```

The character count plus the line count should be equal to the file size as reported by OpenExtensions.

On z/OS, `sample.c` is a perfectly valid name for a sequential data set, whereas `john/sample.c` is not a valid name for a z/OS sequential data set. Thus, if the word contains a forward slash, it is taken to be an OpenExtensions path. Clearly, the full path from the root contains a leading forward slash and will always be interpreted as a reference to an OpenExtensions file. You can always construct a path from the current working directory that contains a forward slash by prefixing `./`, which makes the path explicitly relative to the current working directory:

Figure 55. Reading OpenExtensions Files on z/OS

```
pipe < ./sample.c | count characters lines | console
pipe < /u/john/sample.c | count characters lines | console
```

OpenExtensions file names may contain blanks. To support this, *CMS Pipelines* supports enclosing the path in quotes. On CMS, you can use single quotes or double quotes, as you find most convenient, but on z/OS you must use double quotes, because single quotes denote a fully qualified data set name:

Figure 56. Writing File with Blank

```
pipe literal a blank file|> "a blank file"
```

CMS Pipelines also provides device drivers to read and write binary files in an OpenExtensions file system; refer to *hfs*. See also *hfsdirectory*, *hfsquery*, *hfsstate*, and *hfsxecute*.

Libraries

CMS Pipelines can write information about the contents of a partitioned data set into the pipeline (that is, information from the directory); and it can write the contents of specified members into the pipeline.

listpds writes an output record for each member of a partitioned data set. On CMS, the file name and file type of the library are specified; on TSO, the data set name or DDNAME is specified.

Figure 57. Reading a PDS Directory into the Pipeline on CMS

```

pipe listpds pipident maclib | console
▶describe"*****"
▶PGMID  "*****"
▶CALL   "*****"
▶Ready;

```

The first eight bytes of each output record contain the member name; the remainder of the record is undefined as far as *CMS Pipelines* is concerned. The double quotes represent unprintable binary data.

members reads specified members from a partitioned data set. It can read the names of members to process from its input as well as processing the members specified on its parameter list. This example shows reading a member of a z/OS partitioned data set:

Figure 58. Reading Members of a PDS into the Pipeline

```

pipe literal j | members dd=sysexec | console
▶/* J EXEC */ parse arg file
▶address link
▶'PIPE <' file '|menucl /'file'/ edit'
▶exit rc
▶READY

```

This example shows that *members* can read the member list from its input.

Typing on the Terminal

Another device driver you often need reads from and writes to the console of your virtual machine (your terminal). Like *disk*, it reads when first in the pipeline, and writes in other positions. Figure 59 shows how *disk* and *console* are combined to make the equivalent of the CMS TYPE command.³

Figure 59. TYPE

```

pipe disk small exec | console
▶/* This is a small Exec */
▶signal on novalue
▶say 'Bye...'
▶Ready;

```

Figure 60. Simplistic TYPE Command

```

/* type command */
'pipe disk' arg(1) '|console'
exit RC

```

You can combine device drivers to copy a data stream to several devices or files. In this example, the data are copied to the console as well as to the file “A B” on disk C:

³ This file is also used as the input file in the *xlate* examples in “Translate Characters” on page 44.

Using Device Drivers

Figure 61. Strange TYPE Command

```
/* strange type command */  
'pipe disk' arg(1) '|console|disk a b c'  
exit RC
```

console reads lines you type on the terminal when it is first in the pipeline specification. *console* stops reading when you enter a null line (just hit enter); this line is discarded.

Note: z/OS users should note that *console* reads and writes to the log on terminal; CMS is indeed a “master console operator” application in the sense that it is the program that is IPLed in the virtual machine. On TSO, *console* cannot access a z/OS console; nor does it issue WTO macros to the master console (it uses route code 11 to write to the programmer when the PIPE command is invoked directly from JCL). Use the synonym *terminal* if you find that a more appropriate name.

Injecting Data into the Pipeline

Generate a record inside the pipeline with *literal*. It writes its arguments as the first record and then shorts itself out to copy any input to the output without modification. Thus, a cascade of *literal* stages generates records in the reverse order of the stages:

Figure 62. literal Example

```
pipe literal first|literal second | console  
▶second  
▶first  
▶Ready;
```

Console Stack (External Data Queue)

Use *stack* to put lines on the console stack; an option specifies whether the lines are put at the front (LIFO) or at the end of the queue (FIFO).

There are two ways to read from the console stack when the device driver is first in the pipeline. Use *console* to read lines until a null line is read; use *stack* to read as many lines as there are on the stack.

Using Virtual Unit Record Devices (VM/CMS)

reader reads files in your virtual reader. By default, it reads the first file it can find in your reader. Use the keyword FILE to read a specific file, or put the file up front with the CP command ORDER. Figure 63 on page 35 shows how to read a SPOOL file and type it on the console.

Figure 63. Unfiltered TYPE

```

/* peek a file */
parse arg sfid .
address command
'CP CLOSE RDR'
'CP SP RDR HOLD NOCONT'
'PIPE reader file' sfid '|spec 2-* 1|console'
'CP SP RDR NOHOLD'
exit RC

```

The first word of the command (PIPE) is upper case because the default command environment has been set to COMMAND, which makes the case of CMS commands important. *spec* removes the carriage control character from the beginning of each record. Further note that records that have carriage control X'03' (no operation) are included in the data typed. The first line is the tag of the SPOOL file.

Figure 64 shows how to print a file already containing machine control characters, for instance SCRIPT output:

Figure 64. Print a File with Machine Carriage Control

```

/* Prints upright on A4 on the 3800 on MVS                                     */
signal on novalue
parse arg fn ft fm .
ft=word(ft '3800', 1)
address command
'IDENTIFY(LIFO'
parse pull . . node .
If left(node,5),='CPHVM'
  Then
  Do
    say 'Modify 3800 EXEC with your own',
        'SPOOL and TAG info.'
    exit 12
  end
'CP SP E RSCS NOCONT PURGE NOHOLD CLASS A',
'DIST XAIXB269 FCB S8',
'FORM STD. CHAR IT12 IB12'
'CP TAG DEV E CPHMVS1 SYSTEM 50 SYSOUT=4 OPTCD=J'
'PIPE',
'| <' fn ft fm,                               /* Read file           */
'| unpack',                                     /* In case packed      */
'| xlate 3-* c0 8b d0 9b',                     /* Curlies             */
'| printmc'                                     /* Print               */
r=RC
'CP CLOSE E NAME' fn ft
exit r

```

Lines read by *reader* can be written back to SPOOL with *printmc* without further processing if the SPOOL file is a printer file:

Using Device Drivers

Figure 65. Naive 1 8p7

```
/* Copy Reader File to Printer */  
'pipe reader|printmc'
```

The pipeline specification in Figure 65 copies the reader file to a copy on the printer (if all CCW operation codes are valid for the output device), except that the tag of the reader file appears as an additional no operation record in the printer SPOOL file. (See Figure 166 on page 79 for a command that retains the tag.)

Figure 66 shows how to punch a file:

Figure 66. Simplistic Command to Send a File.

```
/* Simplistic sendfile */  
arg node user file  
call diag 8, 'SP D RSCS PURGE NOHOLD NOCONT CL A'  
call diag 8, 'TAG DEV D' node user  
'pipe disk' file '|' chop 80 | punch'  
call diag 8, 'CLOSE D NAME' subword(file, 1, 2)  
exit RC
```

MVS SPOOL

In a batch job, *TSO Pipelines* can read a SYSIN data set by specifying its DDNAME to <. But *TSO Pipelines* is unable to read data sets that have been sent with XMIT to you directly from SPOOL, because the underlying interface to read the SPOOL file requires the task to be authorised.

You can create a SYSOUT data set in two ways. You can allocate the data set and use DDNAME= with >; or you can use *sysout* to allocate the data set dynamically.

For those with a CMS bent, *printmc* and *punch* are synonyms for *sysout*. *printmc* expects the input records to have machine carriage control (like RECFM=VM); whereas *punch* assumes that no carriage control is present (like RECFM=V).

Accessing Variables

CMS Pipelines can access the variable pools in REXX, EXEC2, and CLIST programs. We recommend that you use REXX programs and on TSO issue the PIPE command by Address Attach.

Use *stem* to read and write variables in the program that calls *CMS Pipelines*. As with EXECIO, <stem>0 is set to the count of records and <stem>n (where n is a positive number) contains the nth individual record. Figure 67 shows how to load a file into a stemmed array beginning with file.1.

Figure 67. Loading a File into a Stemmed Array

```
/* Load file into stem */  
'pipe < some file | stem file.'
```

stem is handy to run subroutines as pipelines without going via the stack or a file. Figure 68 on page 37 shows how to sort the contents of REXX variables. The count of records in the array with stem *unsorted*. must be stored in the variable *unsorted.0*

before running the pipeline. The period after the stem name indicates the use of a stemmed array. No period is added by *stem*; this means that *stem* can also be used with EXEC2 or CLIST.

Figure 68. *stem* Example

```
/* Sorting the contents of stemmed array */
address command 'PIPE',
  'stem unsorted. | sort | stem sorted.'
```

It is possible to read and write the same variables in a pipeline with two *stem* stages that refer to the same stem. This is safe as long as there is a buffering stage (for instance, *sort*) or no records are added to the file in the pipeline, but it is better to be safe and write to a different stem unless the file is so large that two copies cannot fit in storage.

The device driver *var* reads the contents of a single variable into the pipeline when it is first in a pipeline. When *var* is not a first stage, it loads the first record into the variable and copies all input to any following stage; the variable is dropped if there is no input.

Figure 69. Using *var*

```
/* convert from 8-byte floating point */
'PIPE var cpu2busy | spec 1-* c2f | var cpu2busy'
```

The filter *spec* will be described later; as used here, it converts the record from the internal IBM System/390* hexadecimal floating point representation (eight bytes) to a character string that contains the number in scientific notation. After conversion the contents of the variable can be processed by REXX.

Refer to *varset* for a way to set many variables that are not a stemmed array that has consecutive numeric subscripts. *varfetch* reads variables from the REXX environment; the names of the variables are specified in *varfetch*'s input. *vardrop* drops variables.

Using Device Drivers to Read Data into the Pipeline Downstream

If your REXX program has two stemmed arrays that you wish to sort into one, you cannot use a cascade of *stem* stages to read the variables because the second *stem* would not be a first stage and thus it would replace the second array with the contents of the first one. Though the general solution is to use multistream pipelines, two built-in control stages, *preface* and *append*, let you run a device driver as a first stage somewhere downstream in the pipeline.

Figure 70. Using *append*

```
/* Sort two stemmed arrays into one */
'PIPE',
  '| stem first.',
  '| append stem second.',
  '| sort',
  '| stem sorted.'
```

The first *stem* in Figure 70 reads the stemmed array as described in “Accessing Variables” on page 36. *append* copies the primary input stream to the primary output stream and then runs the argument stage, connected to the primary output stream. The input to the argument stage is not connected; it is a first stage and does read the stemmed array.

Using Device Drivers

preface runs the argument stage before it copies the primary input stream to the primary output stream.

preface and *append* run the arguments in the REXX environment in effect when they are invoked irrespective of the number of REXX programs in the pipeline.

You can also use < with *append* to concatenate two files:

Figure 71. Using *append* with Files

```
/* Catenate two CMS files */
arg fn1 ft1 fm1 fn2 ft2 fm2 fn3 ft3 fm3 .
address command
'PIPE',
  '| <' fn1 ft1 fm1,
  '| append <' fn2 ft2 fm2,
  '| >' fn3 ft3 fm3
exit RC
```

```
/* Catenate two MVS files */
arg fn1 fn2 fn3 .
address link
'PIPE',
  '| <' fn1,
  '| append <' fn2,
  '| >' fn3
exit RC
```

Another Way to Read a File

If the names of the files are already in the pipeline, you can use *getfiles* to read the contents of the files named in its input records. You can also use *literal* to put the names of the files into the pipeline if they are not there already. Note the order of the two *literal* stages in the next example; the record from the last one arrives first at the following stage. The last example shows how to concatenate the contents of a variable number of files on z/OS. The *split* stage makes one record for each blank-delimited word in the argument string.

Figure 72 (Page 1 of 2). Reading Two or More Files into the Pipeline

```
/* Catenate two CMS files */
arg fn1 ft1 fm1 fn2 ft2 fm2 fn3 ft3 fm3 .
address command
'PIPE',
  '| literal' fn2 ft2 fm2,
  '| literal' fn1 ft1 fm1,
  '| getfiles',
  '| >' fn3 ft3 fm3
exit RC
```

Figure 72 (Page 2 of 2). Reading Two or More Files into the Pipeline

```

/* Catenate two MVS files */
arg fn1 fn2 fn3 .
address command
'PIPE',
  '| literal' fn2,
  '| literal' fn1,
  '| getfiles',
  '| >' fn3
exit RC

```

```

/* Catenate many MVS files */
arg output rest
address command
'PIPE',
  '| literal' rest,
  '| split',
  '| getfiles',
  '| >' output
exit RC

```

Issuing Commands

Several device drivers issue commands and provide the response as their output, a line at a time. These device drivers issue the argument string, if any, as the initial command; the primary input stream is then read and a command is issued for each line.

Other device drivers issue commands without trapping the response; these are useful to invoke programs that use full screen mode, for instance XEDIT.

If the primary output stream from *cp*, *command*, *cms*, or *tso*, is not connected, the output from the command is discarded. Thus, this allows a way to issue commands and suppress any error messages that might otherwise have been issued.

When the host command interfaces are used without a secondary output stream, they *aggregate* the return codes from the individual commands and provide this aggregate as their return code. CP return codes are zero or positive; a return code of 1 indicates an unknown CP command; for other return codes, the aggregate is the maximum return code. For CMS, if any return code is negative, the aggregate of the return codes is the minimum return code; otherwise the aggregate return code is the maximum of the return codes.

The built-in programs to issue host commands support a secondary output stream. When the secondary output stream is defined, the program writes the return code received on a command to this stream after the output from the command (or the command itself) has been written to the primary output stream. The return codes can be aggregated by passing them to *aggrc*. When the secondary output stream is defined, the return code from the host command interface is zero unless the program itself detects an error. You cannot specify an initial command as the argument when the secondary output stream is defined.

Issuing Commands

CP

The device driver *cp* sends commands to the Control Program (CP) and writes the response to the pipeline. If an argument string is present, it is issued first; then the primary input stream is read and issued.

Figure 73. Example of the CP Device driver

```
pipe cp q files | console
▶FILES: NO RDR, NO PRT, NO PUN
▶Ready;

pipe cp Q files | console
▶Invalid option - files
▶Ready(00003);

pipe literal Q 00C | cp Q FILES | console
▶FILES: NO RDR, NO PRT, NO PUN
▶RDR 000C CL * NOCONT NOHOLD EOF READY
▶ 000C 2540 CLOSED NOKEEP NORESCAN SUBCHANNEL = 0001
▶Ready;
```

CP command names are in upper case; giving CP a command whose name is in lower case results in return code 1 (unknown CP command). To help you, *cp* inspects the first word of each command before it is issued; if the word is all upper case, *cp* issues the command as you have written it, possibly with mixed case arguments. If the first word of the command is completely or partly in lower case, *cp* translates the complete command to upper case before issuing it to CP. This is why the first line in Figure 73 shows the response to the QUERY command; the second example is interpreted by CP as a request to look for the user logged in as “files” in lower case. The last example shows how to issue multiple commands.

It is seldom useful to cascade *cp* device drivers because the output of the first *cp* device driver would be interpreted as commands by the second instance of *cp*, and you would most likely just see a return code of 1 indicating that the line is not a valid CP command. However, the response to a CP command is often used to build another CP command; Figure 74 shows how to close a punch and put the resulting SPOOL file first in the reader queue.

Figure 74. Building a Command from a Response

```
cp spool d to *
pipe < profile exec | punch
pipe cp close d | spec /order rdr / 1 10.4 next | cp
```

For a QUERY command, the response buffer is extended automatically to accommodate the length of the reply. For example, you need not worry about the number of reader files in your reader when you process the response to QUERY RDR * ALL. The default length of the response buffer is 8K for commands other than QUERY; put a number as the first argument to *cp* to use a different size for the buffer. The number specifies the number of bytes to allocate to the buffer to make it larger (or indeed smaller) than the default 8K buffer.

CMS

cms and *command* issue CMS commands and intercept the response normally written to the terminal. *cms* is recommended for casual work because it issues commands with the search order you are used to when you type CMS commands on your terminal. This is equivalent to Address CMS in REXX. Figure 75 shows an example.

Figure 75. CMS Example

```
pipe cms query impcp | xlate lower | console
▶impcp      = on
▶Ready;
```

CMS forwards unrecognised commands to CP. Thus, *cms* can be used to issue CP commands, but the response is written to your terminal by CP; use *cp* to issue CP commands when you wish to process the response. On the other hand, use *subcom* CMS to issue CMS commands without trapping the CMS response.

command issues the command in the same way that Address COMMAND in REXX does; the argument string and input lines to *command* should be upper case unless you wish to manipulate objects with names in mixed case. Figure 76 shows how to create and erase a file with a mixed case name.

Figure 76. command Examples

```
pipe command RENAME LOAD MAP A load = =
▶Ready;

listfile * map
▶load      MAP      A5
▶Ready;

erase load map
▶DMSERS002E File LOAD MAP not found
▶Ready(00028);

pipe command ERASE load MAP A
▶Ready;

listfile * map
▶DMSLST002E File not found
▶Ready(00028);
```

TSO

The *command* device driver issues TSO commands without trapping the command response. It produces no output on the primary output stream. The *tso* device driver issues TSO commands while trapping the response (to the extent that the REXX function OUTTRAP can trap a response).

Issuing Commands

Figure 77. Issuing a TSO Command

```
pipe command time | count lines | console
▶TIME-05:49:19 PM. CPU-00:00:00 SERVICE-6297 SESSION-00:49:33 OCTOBER 14,1992
▶0
▶READY

pipe tso time | count lines | console
▶1
▶READY
```

The first command in the example shows that the response is written directly to the terminal by TSO. The *count* stage does not receive any records from *command* and thus writes 0 to its output. In the second example, the response is indeed intercepted; we do not see it on the terminal. The number 1 indicates that one line was written into the pipeline where it was counted by *count*.

Subcommand Environments

subcom directs the commands to a specified subcommand environment and copies the command to the output; in general, it is not possible to trap responses from subcommand environments. It is unspecified which subcommand environments are available; some environments on CMS may not support standard CMS parameter lists and may cause CMS failures if invoked with standard parameter lists.

Use *subcom* CMS to issue CMS commands without intercepting console output.

The pipeline specification in Figure 78 issues the STATE command for each file mentioned in a file with file type FILELIST, issuing error messages for files that do not exist.

Figure 78. Using *subcom* to Issue CMS Commands

```
/* Ensure all files in a filelist are present */
parse arg fn .
'PIPE <' fn 'filelist|spec ,STATE, 1 w1.3 nw|subcom cms'
exit RC
```

change prefixes the command to the file identifier.

On TSO, the TSO subcommand environment issues TSO commands:

Figure 79. Yet another Way to Issue a TSO Command

```
pipe subcom tso time | count lines | console
▶TIME-06:51:35 PM. CPU-00:00:01 SERVICE-15494 SESSION-01:51:49 OCTOBER 14,1992
▶0
▶READY
```

The example above shows that the command specified as the argument string is not copied to the output; the response is displayed on the terminal, and the count of lines written into the pipeline is zero.

Obtaining CP Messages and other Console Output

This is a specialist item; skip to the next section unless you are interested in programmable operators.

starmsg connects to the message system service provided by CP to intercept console output. Each output line from *starmsg* has a 16-byte prefix, which contains the message class and the name of the originating virtual machine; this is followed by the message or response from CP.

starmsg operates differently when it is first in a pipeline and when it is not first in a pipeline. When it is not first in a pipeline, it will terminate when it reaches end-of-file on its input; use this form to implement clients. When it is first in a pipeline, it will continue waiting for messages until it is terminated by a command or its output is severed; use this form to implement servers.

When *starmsg* is not a first stage, it issues each input record as a CMS command and terminates when it reaches end-of-file. To issue a single command and trap the response, both from CP and CMS:

Figure 80. Issuing a Single Command, Trapped

```
set cpconio iucv
set vmconio iucv
pipe literal release z (det | starmsg | > release response a
set cpconio off
set vmconio off
```

If it is present in the argument, a command is issued immediately after *starmsg* is connected to the system service; this ensures that all output from a command can be trapped. *starmsg* cannot determine when the command is complete; you must make *starmsg* stop somehow if it is first in the pipeline.

When a command ends with a message, for instance “Command complete”, it may be possible to use *tolabel* to stop at this point; the line is discarded. Note that no CMS ready message is issued since the PIPE command is still running; use the Say REXX instruction at the end of a command procedure to write a line that can be used to stop processing. Figure 81 on page 44 shows an example that invokes a command procedure to issue both CP and CMS commands; the response is stored in a file. The command “complex” issues a complex set of CP and CMS commands; it writes “Done?” to the terminal when complete.

Using Filters

Figure 81. Connecting to the Message Service

```
/* Run command in general storing response */
Address command
'CP SET MSG OFF'
'CP SET CPCONIO IUCV'
'CP SET VMCONIO IUCV'
'PIPE',
  'starmsg complex |',
  'tolabel' right(5, 8, 0) || left(userid(), 8)'Done?|',
  '> gen data a'
r=RC
'CP SET MSG ON'
'CP SET CPCONIO OFF'
'CP SET VMCONIO OFF'
exit r
```

starmsg sets up an immediate command (HMSG by default); you can issue this command to stop the stage. Another way to stop an asynchronous pipeline is to issue the command PIPMOD STOP to CMS; you can do that with an immediate command or from a filter written in REXX. This terminates all stages waiting for an external event.

Using Filters

Programs that process data in the pipeline without reference to a host interface are called filters. These functions are typical examples of tasks performed by filters:

- Translate characters, mapping one character to another.
- Count characters, words, and lines.
- Edit the record to rearrange its contents.
- Change the record format and transform between CMS and formats used in other operating systems.
- Select records. You can select records that start with a given string in the same way as the FIND XEDIT subcommand, or ALL XEDIT subcommand. Other filters emit the records that do not match rather than the matching ones.

There are others; *sort* is a distinguished example of the remaining filters.

Translate Characters

xlate replaces each character with another one based on the mapping in a translate table. This is useful, for instance, to change the collating sequence or to blank out unwanted delimiter characters. Translation can be restricted to specified input ranges. The translate table is built by modification to an initial table, the neutral one by default; other initial tables are selected by keyword.

Figure 82 shows how to translate the complete record to upper case.

Figure 82. Hello, world! Pipeline

```
pipe literal Hello, world|xlate upper|console
▶HELLO, WORLD
▶Ready;
```

The following *xlate* examples operate on the file shown in Figure 59 on page 33.

To translate selected positions only:

<i>Figure 83. Translating Selected Positions to Upper Case</i>
<pre> pipe disk small exec xlate (10.5 *-5) upper console ▶/* THis iS A Small Exec */ ▶SIGNAL on NOVALue ▶SAY 'Bye...' ▶Ready; </pre>

Figure 83 shows how to specify two column ranges. Characters within the specified ranges are translated to upper case. Note that ranges may be written in any order and that an asterisk (*) identifies the beginning or the end of the record, as appropriate. A hyphen (-) separates the begin and end column of a range; use a period to append a column count to the begin column number. To reverse the case:

<i>Figure 84. Reverse Translation</i>
<pre> pipe disk small exec xlate upper A-Z a-z console ▶/* tHIS IS A SMALL eXEC */ ▶SIGNAL ON NOVALUE ▶SAY 'bYE...' ▶Ready; </pre>

The opposite of *xlate* UPPER is shown in Figure 85.

<i>Figure 85. xlate LOWER</i>
<pre> pipe disk small exec xlate lower console ▶/* this is a small exec */ ▶signal on novalue ▶say 'bye...' ▶Ready; </pre>

Translation is performed in the order column ranges are written. All or part of a record can be translated more than once; this is noticeable when the translation has no closure:

<i>Figure 86. Double Translation</i>
<pre> pipe disk small exec xlate (1-* 1.10) upper A-Z a-z console ▶/* This is A SMALL eXEC */ ▶signal on NOVALUE ▶say 'Bye...' ▶Ready; </pre>

Here the first 10 columns are translated twice, and the letters go back to their original case.

The neutral translate table is used when no keyword is specified, as in this example removing special characters:

Using Filters

Figure 87. Remove Special Characters

```
pipe disk small exec | xlate 1-* 00-80 40 | console
▶ This is a small Exec
▶signal on novalue
▶say Bye
▶Ready;
```

It is a good idea to use an explicit input range when the default translate table is used. This ensures that the first item of the translation specification is not taken to be a single column range:

Figure 88. Example of xlate Error

```
pipe disk small exec|xlate 01-80 40 | console
▶Odd number of translate pairs
▶... Issued from stage 2 of pipeline 1
▶... Running "xlate 01-80 40"
▶Ready(00053);
```

The first token is interpreted as a desire to translate the contents of columns 1 to 80 inclusive, though the intent was to translate that range to blank characters.

Figure 89. Caesar Cipher

```
pipe < small exec|xlate a-y b-z i j r s z a | console
▶/* Tijt jt b tnbmm Eyfd */
▶tjhobm po opwbmvf
▶tbz 'Bzf...'
▶Ready;
```

Figure 89 shows a transliteration of the lower case letters. Upper case letters are not affected. The simpler specification a-y b-z z a is not used because the “holes” in the EBCDIC collating sequence would turn i and r into characters that are not letters.

Counting

count counts the amount of data in the input stream. It counts characters, words, and lines. The result is a single record, which is written to an output stream. The result is written to the secondary output stream, if defined. Input lines are copied to the primary output stream when the secondary output stream is defined.

Figure 90. count Example

```
pipe < pipeug script | count characters words | console
▶108400 17989
▶Ready;

pipe < pipeug script | xlate 1-* 40-7f blank | ...
... count characters words lines | console
▶108400 19098 4282
▶Ready;
```

The *xlate* stage turns all special characters into blanks. The counts are always in the order characters, words, and lines irrespective of the order in which the options are specified.

Editing and Conversion

Two filters, *change* and *spec* change the contents of a record. *change* replaces occurrences of one string with another one. *spec* reorders fields, converts data, and inserts literal data and line numbers.

change

change works like the CHANGE XEDIT subcommand; as in Figure 78 on page 42, it is often used to put a literal in front of each line.

The default, however, is to change all occurrences; specify the maximum number of substitutions after the change string specification.

Figure 91. *change*

```
pipe disk small exec | change /l/*****/ | console
▶/* This is a sma***** Exec */
▶signa***** on nova*****ue
▶say 'Bye...'
```

```
▶Ready;
```

```
pipe disk small exec | change /l// | console
▶/* This is a sma Exec */
▶signal on novalue
▶say 'Bye...'
```

```
▶Ready;
```

Figure 91 shows how to change all occurrences of the letter “l” to five asterisks and how to remove “l”. You see that the default scope is the complete record.

Figure 92. *More change*

```
pipe disk small exec | change /l// 1 | console
▶/* This is a smal Exec */
▶signa on novalue
▶say 'Bye...'
```

```
▶Ready;
```

```
pipe disk small exec | change 10-* /l// | console
▶/* This is a sma Exec */
▶signal on novaue
▶say 'Bye...'
```

```
▶Ready;
```

In Figure 92, only the first l is changed; a range is specified in the second part. This is similar to the ZONE XEDIT subcommand setting, but you can have more than one range in parentheses (not shown).

change ANYCASE supports mixed case change strings. That is, the twenty-six letters are compared irrespective of their case when *change* looks for a string to replace. If the first string contains no letters or contains one or more upper case letters, the second string replaces occurrences of the first string without further change. When the first string is in lower case, *change* tries to preserve the case of the string being replaced:

Using Filters

Figure 93. *change ANYCASE*

```
pipe literal The flower and the bee | change anycase /the/any/ | console
▶Any flower and any bee
▶Ready;
```

spec

spec has evolved into a program that is rich in function; so much that it has been given two separate chapters in this book. (Chapter 16, “*spec* Tutorial” on page 166 and Chapter 24, “*spec* Reference” on page 719.) Here we show the original simple form of *spec*, which is still useful in its own right.

As used originally, *spec* builds an output record for each input record. The output record contains one or more fields, which can contain literal data or data from a field of the input record. The specification list (from which *spec* got its name) consists of pairs of input and output specifications:

```
... | spec <input-1> <output-1> <input-2> <output-2> | ...
```

For each input record, *spec* goes through the specification list and performs the actions specified. By the time it reaches the end of the list, it has built the output record.

There is no arbitrary upper limit on the number of items in a specification list.

To put the contents of the input record into the output record at column 11:

Figure 94. *Shifting the Input Record*

```
pipe literal abcd | spec 1-* 11 | console
▶          abcd
▶Ready;
```

This specification list contains one pair of input and outputs. The input range (1-*) represents the entire input record beginning in column 1 and extending to infinity (the asterisk is idiomatic for the largest integer that the computer can handle). *spec* does not pad the input record; it takes the shorter of the actual record and the range specified. It puts the input field into the output record starting at column 11 and fills the unspecified part of the output record with blanks. Again, it does not pad the output record beyond the area filled by the input field.

To prefix the contents of each input record with a literal that is repeated in all output records:

Figure 95. *Prefixing a Literal*

```
pipe literal abc | spec /Input: / 1 1-* next | console
▶Input: abc
▶Ready;
```

In this example, the literal field, which is specified between the two slashes, is inserted into the beginning of the output record followed by the contents of the input record (1-*). The keyword NEXT specifies that the field should be appended to the rightmost character inserted into the output record so far.

Literal fields are delimited by a special character, which is traditionally a forward slash (/). The delimiter character is deleted when the field is inserted in the output record.

For example, *spec* can be used to prefix a record with an identification of its origin as in Figure 96. The file name, type, mode, and record number are prefixed to each input record.

This is done by putting three literal fields containing the file name, file type, and file mode into columns 1, 10, and 20, respectively; the record number into columns 25 through 34; and finally appending the contents of the input record in column 41 onward:

Figure 96. Using *spec* to Identify Records of a File

```
/* Prefix Identification */
'pipe',
  'disk' fn ft fm '|',
  'spec /'fn'/ 1 /'ft'/ 10 /'fm'/ 20 number 25 1-* 41|...
```

The specification list in Figure 96 contains five items:

1. A literal to insert the file name into column 1 and onward.
2. A literal to insert the file type into column 10 and onward. If the file name is shorter than nine characters (which one would expect), the slack is filled with blanks. Should the variable *fn* contain more than ten characters, the eleventh character and onward will be overlaid by the file type.
3. A literal to insert the file mode into column 20 and onward.
4. A reference to the current record number (NUMBER) to insert this into columns 25 through 34. The record number is an internal variable, which *spec* maintains. It is incremented each time *spec* returns to the beginning of the specification list.
5. A reference to the entire input record, which is inserted into column 41.

A forward slash (/) is used to delimit the literal fields because it is not a valid character in a file name and thus should not occur in the literal data itself (unless the files are OpenExtensions files). A hyphen in an input range indicates that the field is specified as the beginning and ending column. You can use a period instead of a hyphen to specify the number of columns in a field rather than its ending column.

A field of the input record can be specified as beginning or ending relative to the end of the record rather than the beginning. (Or both beginning and ending relative to the end of the record.) Put a minus sign in front of the column number to make it relative to the end of the record. You must specify a beginning and ending column number separated by a semicolon when using this notation. There is no provision for a column count in this format.

Figure 97. *spec* Relative to the End of a Record

```
pipe literal abcdefg ab abc | split | spec 2;-2 1 | console
▶bcdef
▶
▶b
▶Ready;
```

Using Filters

Figure 97 shows a target that is relative both to the beginning and end of a record; the first and last character are removed from each record. A record with two characters or less becomes a null record, because *spec* ignores fields that have zero or negative lengths. When the field is ignored, the output record is not padded to the position where the field would have gone if it were not null.

<pre>pipe < small exec change 15-* //*****/ console ▶/* This is a s*****mall Exec */ ▶signal on nova*****lue ▶say 'Bye... ' ▶Ready;</pre>
<pre>pipe < small exec spec 1.14 1 ,*****, next 15-* next console ▶/* This is a s*****mall Exec */ ▶signal on nova*****lue ▶say 'Bye...'***** ▶Ready;</pre>
<pre>pipe < small exec spec 1.14 1 ,*****, 15 15-* next console ▶/* This is a s*****mall Exec */ ▶signal on nova*****lue ▶say 'Bye...' ***** ▶Ready;</pre>

Finally, Figure 98 shows how to replace the null string with five asterisks. Note the difference between *spec* and *change* in the last record. *spec* inserts the string in the last record as well; *change* does not because the last record does not extend to the column range.

Specifying Input Ranges

You have seen in the previous sections several ways to bring a filter to operate on part of the input record. In the syntax diagrams in the reference part of this book, you will see such expressions referred to as *inputRange*.

The specification of an input range grew from the simplistic range you can specify with the COPYFILE command. With COPYFILE (SPEC, you could specify only the first and the last column of a range. Thus, COPYFILE ranges are always specified relative to the beginning of the record.

In contrast, *CMS Pipelines* allows you to specify an input range relative to either the beginning or to the end of the record. A range that is relative to the end of the record is specified with a leading hyphen (-). And you can specify more than just a column count:

- Blank-delimited words.
- Tab-delimited fields.

If you prefix the range with the keyword WORD, the number(s) specify words, which are separated by one or more blanks.

Figure 99. Selecting Words

```
pipe literal Such a fine day. | spec word 3 1 | console
▶fine
▶Ready;
pipe literal Such a rainy day. | change word -2 /rai/sun/ | console
▶Such a sunny day.
▶Ready;
```

You can even specify a different word separator character if you prefix the WORD keyword with the keyword WORDSEPARATOR and then specify which character you want used to delimit words. This can be useful when you want to treat a run of delimiter characters as a single delimiter:

Figure 100. Specifying a Word Separator

```
pipe literal a***b***c | spec wordsep * word 2 1 | console
▶b
▶Ready;
```

Tab-delimited fields are very similar to blank-delimited words; the difference is that two consecutive field separators specify a null field. By default, the field separator is X'05', the horizontal tab character. You will probably change the field separator more often than you change the word separator:

Figure 101. Specifying a Field and a Field Separator

```
pipe literal a***b***c | spec fieldsep * field 4 1 | console
▶b
▶Ready;
```

This may be complex already, but you will soon find yourself wishing to extract the second character of the third word:

Figure 102. Selecting a Substring of an Input Range

```
pipe literal What a rum fellow. | spec substr 2 of word 3 1 | console
▶u
▶Ready;
```

You can even take the substring of a substring, and so on *ad infinitum*. (See Figure 145 on page 69)

Selecting Records

Several built-in programs select records based on the contents of the records and the arguments specified. To select records based on their contents, there are filters that work like:

- REXX built-in functions Abbrev and Verify.
- XEDIT subcommands Find, NFind, and Locate
- COPYFILE options FRlabel and T0label.

Using Filters

Overview

take, *drop*, *flabel*, and *tolabel* partition the data stream by deleting or copying records to or from a specified position in the data stream. *between*, *inside*, *notin*, *outside*, and *whilelabel* do this for ranges of records based on leading characters.

all, *asmfind*, *asmnfind*, *find*, *nfind*, *locate*, *nlocate*, and *unique* select individual records based on contents.

Four built-in programs require a selection stage as their argument string. These programs (or prefixes) modify the behaviour of the specified selection stage. This concept may be slightly mind-boggling at first, but you will soon see examples of their use.

casei is specified as a prefix to a selection stage to make the selection stage disregard case while it performs its operation, for example to select records that contain a character string irrespective of the case. *zone* is specified as a prefix to a selection stage that does not support a column range otherwise; a specified range of each record is tested rather than the beginning of the record. *casei* and *zone* prefixes can be combined to perform the composite operation; the order of the prefixes does not influence which records are selected.

Be sure to use the selection stage's own facilities for caseless operation and input ranges, if it supports such; this will be more efficient and you are likely to find more facilities than are offered by *casei* and *zone*.

frtarget and *totarget* are also specified as a prefix to a selection stage. *frtarget* selects records starting with the first one that is selected by the selection stage; *totarget* selects records up to (but not including) the first record selected by the argument stage. Thus, *frtarget* and *totarget* partition the file even if the specified selection stage does not.

All selection filters operate like a railway junction; each record is sent to the primary output stream or the secondary output stream, depending on whether the contents of the record satisfy a condition specified as a parameter to the filter. Though definitely useful (see "Decoding Trees" on page 82), the secondary pipeline is not defined most of the time; records destined for it are then discarded. In this chapter, however, we are only concerned with what happens on the primary pipeline.

Selecting Individual Records

Suppose you wish to find in an ASSEMBLE program all lines that have the operation code PIPERM. "all/ PIPERM /" is an XEDIT subcommand to do this while editing a file. With *CMS Pipelines*, you can select those lines in this way:

Figure 103. *locate* Example

```
/* Find error messages */  
'PIPE disk myprog assemble | locate / PIPERM / | console'
```

This *locate* stage selects all records that somewhere contain the string between the forward slashes (/). You get at least all lines with the operation code PIPERM because an operation code must have at least one blank character on each side.

locate supports only one string; use *all* to select lines that contain one of several strings, or contain several strings.

Figure 104. Using all

```

pipe literal abc def ghi | split | all /b/ ! /i/ | console
▶abc
▶ghi
▶Ready;

```

The exclamation sign specifies that a record is selected if one or both of the strings is contained within it.

find selects records where the leading string is equal to the parameter. As with XEDIT, blank characters in the argument correspond to positions to be ignored during the comparison, and underscores are positions where there must be a blank character to match.

Figure 105. find Example

```

pipe console | find abc | console
a line
abcdefgh
▶abcdefgh

▶Ready;

```

In the example in Figure 105, all records starting with “abc” are selected, as you would expect. Suppose, however, that the parameter had been written with three trailing blank characters. How should *CMS Pipelines* interpret this? Blank characters mean “don’t care” positions; do trailing blank characters make any difference in *find*?

Experiment when wondering about things like this. In the example in Figure 106, no lines were entered with trailing blank characters; the records were as long as what you see.

Figure 106. find With Trailing Blanks

```

pipe console | find abc | console
abcdefgh
▶abcdefgh
abc
abc...
▶abc...

▶Ready;

```

The line with abc only was not echoed; thus, trailing blank characters are significant in the parameter list for *find*.

Use *pick* to select records which satisfy some relation between the contents of a field and a literal, or between two fields in the record. To select records from July 1994 and later (assuming dates in the ISO format):

Figure 107. picking Records after a Date

```

... | pick word 1 >= /19940701/ | ...

```

To discard records where the first two words are equal:

Using Filters

Figure 108. picking Records with a Difference

```
... | pick word 1 != word 2 | ...
```

This also discards null and blank lines, because a missing word is considered to contain no characters; and two null words are considered equal.

pick can (unlike *locate* and *find*) reject records that are exactly equal to some string and select records that contain further data:

Figure 109. Picky pick

```
... | pick 1-* != /Reject these only./ | ...
```

You can use *verify* to select or reject records that contains characters from those specified in a list; for example, to verify that the first word of the record is numeric:

Figure 110. Using verify

```
pipe literal 0644 | literal abcd | verify w1 /0123456789/ | console
▶0644
▶Ready;
```

Since the *literal* stage in produces a record that includes the trailing blank, the *verify* is done only on the first word of the record.

Partitioning the File

The partitioning selection stages divide the input file into two contiguous parts. Thus, they select records up to some particular record or from some particular record.

take copies up to a specified number of records to the output, discarding any further records. *drop* does the opposite: it discards records and copies the balance of the file to the output. Both filters work on the last part of the data stream if instructed by the keyword *LAST*. To see the last lines of a linkage editor output:

Figure 111. take Example

```
pipe disk $$temp lkedit | take last 3 | console
▶ ****NPIPE DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
▶ AUTHORIZATION CODE IS 0.
▶
▶Ready;
```

Use *flabel* to copy records starting with the first one that has a given string in the first positions. This is equivalent to the *COPYFILE* option *FRLABEL*. *tolabel* is the converse operation: it copies records up to but not including the matching one. Both stages match the parameter string verbatim in the case entered; blank characters and underscores mean just that. To see who Tim is (note that the label must be in the correct case and that not finding it gives no diagnostic):

Figure 112. *frlab* Example

```

pipe disk vmsg my | frlabel tim | take 2 | console
▶Ready;

pipe disk vmsg my | frlabel TIM | take 2 | console
▶TIM      TDCSYS3 C33-TCH  HARTMANN
▶**+ *F Tim *N Timothy C. Hartmann
▶Ready;

```

If, on the other hand, you want to see what is immediately before Tim:

Figure 113. *tolab* Example

```

pipe disk vmsg my | tolab TIM | take last 2 | console
▶---- sanjose address
▶**+ *A IBM Corporation;5600 Cottle Rd;San Jose    CA 95193;;USA
▶Ready;

```

Selecting Groups of Records

The *between* family of selection stages works like a combination of *frlabel* and *tolabel* that is run repetitively to obtain records within (or outside) ranges. The argument is a pair of delimited strings, or a delimited string and a number. *between* sends records to the primary stream from (and including) a record with the label specified in the first string up to and including the first occurrence of the second string. It then looks for the next occurrence of the first string again. Records not selected are sent to the secondary output stream, if present. When the second operand is numeric, the number specifies how many records are selected starting from an occurrence of the label defined in the first argument.

Figure 114. *between* Example

```

pipe console | between /a/ /c/ | console
123
aaa
▶aaa
anything
▶anything
centro
▶centro
officio

▶Ready;

```

Figure 115 on page 56 shows the use of *inside* with the second argument a number. Note that an occurrence of the first string within the records being selected is not treated as a recursion. That is, even though there were two records that begin with a, the first record that begins with c terminates the range of selected records.

Using Filters

Figure 115. *inside Example*

```
pipe console | inside /a/ 2 | console
123
a record
the second
▶the second
and a third
▶and a third
the fourth one
the last one

▶Ready;
```

Selecting Unique Records

Removing duplicates is another thing that is useful in general; *unique* removes multiple adjacent occurrences of a record. Use *unique* to remove runs of duplicate records. *sort* UNIQUE is described later; it sorts the file, retaining the first record that contains a particular sort field.

Figure 116. *unique Example*

```
pipe literal a a a b b b a a a c c c | split | unique | console
▶a
▶b
▶a
▶c
▶Ready;
```

Selection stages can be cascaded to fine-tune the set of records selected as shown in Figure 112. As a further example, a CMS file with file type COPY may contain several members. Each member is prefixed by a record that contains “*COPY <name>”. To select one such member:⁴

Figure 117. *Cascaded Selection Stages*

```
pipe disk pipeif copy | unpack | flabel *COPY PIPSTRLB | ...
... drop 1 | tolab *COPY | chop 60 | console
▶          MACRO
▶&L        PIPSTRLB &EXIT=
▶&L        PIPCALL CVTSB,EXIT=&EXIT
▶          MEND
▶Ready;
```

The trailing blank characters on *flabel* and *tolabel* are deliberate. Note that this example fails without *drop*: *tolabel* would match the first record it sees, which would cause no records to be selected.

⁴ The command is printed on two lines because it is longer than the figure width. The two ellipses (...) are not part of the command issued.

Caseless Operation

All built-in programs, for which it is appropriate, support the ANYCASE option to specify that they should ignore the case of the letters a through z; that is, treat “a” and “A” as the same character for purposes of comparison:

Figure 118. Using the ANYCASE Option

```
pipe literal abc def | split | verify /ABCD/ | console
▶Ready;

pipe literal abc def | split | verify anycase /ABCD/ | console
▶abc
▶Ready;
```

In the first pipeline in Figure 118, case is respected and both input records are rejected.

For a selection stage that does not support the ANYCASE option (it would have been written by a user), you can instead use *casei* as a prefix to the selection stage to select records independent of the case of the contents of the record and the case of the search argument:

Figure 119. Using *casei*

```
pipe literal A sentence that has words | split | casei myloc /a/ | ...
... console
▶A
▶that
▶has
▶Ready;
```

Note that there is no stage separator immediately after *casei*.

Refer to “Destructive Testing” on page 84 when your alphabet contains more characters than the twenty-six used by the English. It explains how to use a derivative of the actual record to control the selection.

Splitting, Chopping, and Stripping

The filters *split*, *chop*, and *strip* restrict the size of a record. *strip* can work from both sides; the others work only left to right. They are described together here because they have common syntax and function: they work by scanning the record for a pattern (single character or string); the difference is the processing that follows. They are introduced by showing their default mode of operation. (*chop* needs a little help because its default is to truncate after column 80.)

Using Filters

Figure 120. *strip, split, chop: Default Usage*

```
pipe literal something |strip|spec 1-* 1 /</ next | console
▶something<
▶Ready;

pipe literal a few words |split | console
▶a
▶few
▶words
▶Ready;

pipe literal |spec ,abcdefghij, 75|chop|spec 70-* 1 | console
▶ abcdef
▶Ready;
```

The *split* family of filters matches a string or a single character. For a string, write the keyword `STRING` (which can be abbreviated to `STR`) followed by the data between two occurrences of a delimiter character in the standard CMS way.

Figure 121. *split Family String Target*

```
pipe literal abcaabccabc|strip string ,abc, | console
▶aabcc
▶Ready;

pipe literal abcaabccabcaa|split str ,ab, | console
▶ca
▶cc
▶caa
▶Ready;

pipe literal abcaabccabcaa|chop str /ccab/ | console
▶abcaab
▶Ready;
```

A 1-byte target can be specified in several ways:

- A single character written as such (for example: `z`) or a two-digit hexadecimal representation (for example: `a9`).
- A range of characters, written as `<from>-<to>` or `<from>.<count>`, where `<from>` and `<to>` are characters (1- or 2-byte representation) and `<count>` is a number. A character range wraps from `X'FF'` to `X'00'`.
- An enumerated set of characters identified by the keyword `ANYOF` followed by a delimited string containing the characters.

Figure 122. Character Targets for the *split* Family

```

pipe literal abcaabccabcaa|chop any /c / | console
▶ab
▶Ready;

pipe literal aaaabbbb|strip ,-a | console
▶bbbb
▶Ready;

pipe literal abcaabccabcaa|split a | console
▶bc
▶bcc
▶bc
▶Ready;

```

The pattern is “reversed” with the keyword NOT (or TO in the case of *strip*). When used with a single character pattern, NOT means the complement set with respect to the universe of all 256 values that can be stored in a byte. NOT used with a string pattern means, skip occurrences of the string; the pattern matched is considered to have length 1.

Figure 123. NOT Option on *split* Family

```

pipe literal a word |strip not a | console
▶a
▶Ready;

pipe literal abcaabccabcaa|split not str ,ab, | console
▶ab
▶ab
▶ab
▶Ready;

```

strip further lets you say which side (or both) you want stripped and optionally a maximum count of characters stripped. A record is not discarded by *strip* if all of it has been matched; a null record (having zero bytes) is written.

Figure 124. *strip*: Further Options

```

pipe literal aaaaaaaaaaaaaa|strip leading a 7 | console
▶aaaaaaaaa
▶Ready;

pipe literal |strip trailing|spec 1-* 1 /</ next | console
▶<
▶Ready;

```

The default for *split* is to do it AT the target, which is then removed; use the keywords BEFORE or AFTER to designate that the target should remain and be included in the second or first record, respectively. *chop* truncates before the target by default; use the keyword AFTER to truncate after the target. For both *chop* and *split*, the options BEFORE and AFTER are further modified by a number. This is an adjustment to go past the target when the number is positive (left for BEFORE, right for AFTER); a negative number moves “through” the target in the opposite direction. Note that AFTER is provided as a convenience; n AFTER <pattern> is just a way of expressing m BEFORE <pattern>, where m is

Using Filters

`-n-length(<pattern>)`. This definition sidesteps the question, how long is a string that is not there?

Figure 125. *split and chop: Additional Positioning*

```
pipe literal abcaabccabcaa|split 1 after a | console
▶ab
▶caa
▶b
▶ccab
▶caa
▶Ready;

pipe literal abcaabccabcaa|chop -2 after c | console
▶a
▶Ready;
```

Joining

join puts records together. Specify one less than the number of input records to be used when building an output record. You can add a literal string between records joined.

Figure 126. *join Examples*

```
pipe literal a few words in a sentence | split | join | console
▶afew
▶wordsin
▶asentence
▶Ready;

pipe literal a few words in a sentence | split | join 2 | console
▶afewwords
▶inasentence
▶Ready;

pipe literal a few words in a sentence | split | join 2 /**/ | console
▶a**few**words
▶in**a**sentence
▶Ready;

pipe literal a few words in a sentence | split | join * / / 20 | console
▶a few words in a
▶sentence
▶Ready;
```

The first example in Figure 126 shows the default of joining two records with no added characters; the second one shows the effect of “join 2”; the third one shows adding a string between the records joined. The fourth one shows how to limit the length of the output record. This can be used to flow text if the input records contain words, but it does not provide for flowing of text in general because *join* never splits input records.

join also supports a key field at the beginning of the record. Only records that have the same key are joined.

In general, it is not possible to do the reverse of *split* because it generates a variable number of output records for each input record.

Records from multiple streams are joined by *spec* with SELECT, and *overlay* (possibly after offsetting one of the streams with *spec*).

Use *joincont* to join records when continuation is indicated by the presence (or absence) of a string at the end of a record being continued or at the beginning of the continued line. To join lines using the C convention of suppressing line end (“splicing lines”):

Figure 127. *joincont* Sample

```
pipe literal one two\ three four | split | joincont /\| | console
▶one
▶twothree
▶four
▶Ready;
```

This is too simplistic for real C programs, however, because the line could end in an even number of backward slashes. In this case, the line is not to be spliced. To prevent *joincont* from being fooled by double backward slashes, we turn double backward slashes into something else before joining; and we remember to turn the backward slashes back again. For this to succeed, we need to know that some particular character does not occur in any input record (or we are prepared to accept that such a character is turned into a backward slash):

Figure 128. Production Strength C Line Splicer

```
/* Read C program, splice lines */
'PIPE (name PIPEUG)',
  '|<' fn 'c',
  '|change /\|/ x0000',
  '|joincont /\|',
  '|xlate 1-* 00 \',
  '|...
```

In the example above, double backward slashes are turned into two null characters. Because *change* works from left to right, an odd number of contiguous backward slashes will leave one backward slash at the right (it would not have been so good if it were at the left).

Use *deblock* LINEEND or *deblock* STRING when a particular character or string separates logical records. This deblocking operation is a combination of blocking and deblocking because it joins lines together when the end of line sequence is not at the beginning or end of a record. *deblock* removes the delimiter character or string; it must be reinserted (for instance with *spec* or *change*) if it is to be retained.

Finally, *asmcont* processes an ASSEMBLE file to join all lines of a continued statement into one record. Columns 72 to 80 are discarded, as are columns 1-15 of continued statements.

Changing Record Formats

Having read a file, you may wish to unpack it. The file PIPEODENT COPY is packed; the first example shows what it looks like when you forget to unpack it.

Using Filters

Figure 129. *unpack* Example

```
pipe disk pipodent copy | chop 60 | console
▶ F &«*COPY PGMID : 001: 0 cGBLC -&PGMID,&MODULE7: 002: 0
▶Ready;

pipe disk pipodent copy | unpack | chop 60 | console
▶*COPY PGMID
▶ GBLC &PGMID,&MODULE
▶&PGMID SETC 'PIP'
▶Ready;
```

pack does the reverse of *unpack*. To pack variable record format files with unknown record length, the secondary output stream from *pack* is connected to the secondary input stream to *disk*. This ensures that the header in the first record indicates the correct logical record length for the file; in general, this is only known when the complete input stream has been processed.

When you read from the reader or a tape, what you get is not always unblocked records; often you see blocks in a format peculiar to the program that has made the file as with DISK DUMP, TAPE DUMP, OS record descriptor words, or utilities. And these are often nested. Here is a simple example:

Figure 130. Reading a z/OS Tape in Format V, VB, VS, or VBS

```
/* reads OS V, VB, VS, and VBS tape */
parse arg fileid
'pipe tape | deblock v | disk' fileid
```

Use the option Fixed on *deblock* if the tape is fixed blocked. Sometimes you need more than one deblocking stage to get records completely unwrapped if they are in a Chinese box (for instance a partitioned data set that has been sent in netdata format from z/OS).

Sorting

sort processes files of moderate size that can be held in virtual storage for the duration. You may be able to use *dfsrt* to sort large files.

The *sort* filter reads the file to sort from its input and writes the sorted file to its output; the only options specified on *sort* are the sort fields, which default to the complete record. *sort* normally compares sort fields as binary data; though you can specify the ANYCASE keyword to make it ignore the case in records, this is inefficient for large files. To sort efficiently on a field irrespective of its case you must generate a sort key that has the data folded to the case required. You must also do this if you are dealing with text in a language where the rules for capitalisation are different from English. Remove the key after *sort*:

Figure 131. Generating an Upper case Sort Field

```
... | spec 1-10 1 1-* 11 | xlate 1-10 upper | sort 1-10 | spec 11-* 1 |
```

The translate stage is extended like this to support the Danish collating sequence:

Figure 132. Generating a Danish Upper case Sort Field

```
... | xlate 1-10 upper # ea @ eb $ ec c0 ea 6a eb d0 ec | sort 1.10 |
```

sort reflects the beauty of the pipeline because it only has to sort. Changing the collating sequence is done elsewhere in the pipeline specification, so there is no need for the exits one sees in *sorts* that also have to read and write files.

Note that a range is a single word, unlike the CMS command where you specify the beginning and the ending column of a range as separate words. However, *sort* accepts up to 10 ranges and it is perfectly proper to have a “*sort 1 10*”, but then you are asking for a sort on columns one and ten only.

sort UNIQUE can generate a list of words in a file:

Figure 133. *sort* UNIQUE Example

```
< pipeug script|xlate up 00-80 40|split|sort uniq|count lines|console
▶1721
▶Ready;
```

This is of course naive in the extreme: numbers, examples, DCF control words, and GML tags are also considered words⁵.

collate merges detail records into a master file; *lookup* retrieves records from a master file, based on keys; and *merge* merges records from multiple streams according to a sort key. How to define such streams is described in Chapter 5, “Using Multistream Pipelines” on page 74.

Cascading Filters

As you have seen by now, it is normal to combine several filters to do the job one wants done. One advantage of using *CMS Pipelines* is that you can do this easily.

This section shows how *CMS Pipelines* is normally used by combining several filters. You see examples with more stages than previously, but that is not necessarily the best way to combine filters in real life.

One way is to combine the result of REXX functions that each return part of a pipeline specification. Figure 66 on page 36 shows how to tag and SPOOL a device and punch a file on it. A better approach for the pipeline could be:

```
'pipe disk' file pchprim(node user)
```

Figure 134 on page 64 shows a possible PCHPRIM EXEC.

⁵ See the “programming pearls” column in the May 1985 issue of *The Communications of the ACM* (28:5). Reprinted in Jon Bentley, *Programming Pearls*, Addison-Wesley 1986; ISBN 0-201-10331-1.

Using Many Filters

Figure 134. Using a REXX Function Reference to Generate Part of a Pipeline Specification

```
/* PCHPRIM EXEC */
/* Prime the punch and set up the pipeline */
arg node user .
address command
'IDENTIFY(LIFO'
parse pull . . . . net .
'CP SPOOL D' net 'PURGE NOCONT CL A'
'CP TAG DEV D' node user
return '|chop 80|punch'
```

Figure 135 shows a different, and in many ways better, approach. It is a REXX stage that performs the required CP commands and then redefines itself to punch the data stream.

Figure 135. Using a REXX Filter to Replace Part of a Pipeline Specification

```
/* PCHPRIM REXX */
/* Prime the punch and set up the pipeline */
arg node user .
/* Address CP to issue commands */
address command
'IDENTIFY(LIFO'
parse pull . . . . net .
'CP SPOOL D' net 'PURGE NOCONT CL A'
'CP TAG DEV D' node user
/* revert to the pipeline */
address
'callpipe *:|chop 80|punch|*:'
exit RC
```

The same CP commands are issued in the two examples; the difference is in the way the pipeline specification is issued.

In the first example, the function returns a character string that is made part of the pipeline specification. Thus the stage separator must be a solid vertical bar (or made an argument). Errors are reported (though not shown) by not returning data, which forces a syntax error in the calling REXX program.

The second example runs as a stage; it uses CALLPIPE⁶ in the second last line to replace itself with a new pipeline specification. The pipeline specification is independent of the stage separator specified in the first pipeline and errors can be reported with a return code. (The stages with “*:” are required to show that the new pipeline connects to the existing one.)

⁶ See “Using CALLPIPE to Run a Subroutine Pipeline” on page 103.

Netdata Format

Assume that you have a netdata file in your reader. The *reader* device driver reads the SPOOL file, but the output is not the data set in a format you normally want.

The CCW operation code is in the first position of the record. Records with data you are interested in have X'41' in the first position. The remaining records have X'03' and should be ignored.

A more subtle difference is that CP discards trailing blank characters from the records in the SPOOL system. The physical transmission format does not take this into account and the deblocking stage fails if it gets short records.

The block size is indeed in a transmission header, but it is not required that it be in the first physical record. Thus, the strategy adapted by *deblock* NETDATA is to ignore the problem and insist that the data set be padded to the appropriate block size before the deblock stage. (If CP has saved the original length in the SPOOL file, *reader* finds it and pads the record.)

Now you can recreate a sequential data set with variable record length containing control records and data records. This data stream can be inspected by a stage that redefines itself to either create the desired data set or send the data directly into XEDIT for peek.

This example shows how to peek a file in netdata format:

Figure 136. Example TYPE

```

/* PEEK */
address command 'ERASE PEEKED FILE A'
address command 'CP SPOOL RDR HOLD'
push 'CMS PIPE',
  '| reader', /* Read from spool */
  '| strfind x41', /* retain only data records */
  '| spec 2-* 1.80', /* Discard the CCW opcode; pad */
  '| deblock net', /* Join spanned records */
  '| strnfind xe0', /* Discard control records */
  '| spec 2-* 1', /* Discard the flag byte */
  '| xedit peeked file a' /* Data records to Xedit */
'xedit peeked file a'

```

Here the pipeline specification is pushed so that it is issued when there is an active XEDIT session. However, it is not recommended to stack the complete command; for one thing, it might be longer than XEDIT's truncation limit of 255. Instead, write an XEDIT macro that issues the pipeline command to CMS. Then stack a call to this macro.

To be truly compatible with the default of PEEK, add "take 200|" to the pipeline after *nfind*. This shows only the first 200 records of a large file so that XEDIT does not run out of storage. Of course, what you see is indeed the first 200 records, not the first 200 card images in the transmission data set.

Using Many Filters

Use *deblock* TEXTUNIT to deblock the text units in the control record into separate records (see Figure 137 on page 66 ⁷). The first halfword defines the type of data, the second is the number of fields; and each field is preceded by a halfword length.

Figure 137. *deblock* TEXTUNIT Example

```
pipe < textunit testcase|pad 80|deblock net|find \|deblock textunit| ...
... take 5|spec 1-* 1 /</ next 1-* c2x 25 | console
▶ â      &<          0042000100020050
▶      CPHVM1<      101100010006C3D7C8E5D4F1
▶      JOHN<        101200010004D1D6C8D5
▶      CPHVM1<      100100010006C3D7C8E5D4F1
▶      JOHN<        100200010004D1D6C8D5
▶Ready;
```

Records starting with X'E0' (the backward slash, \) are control records, which are selected. Text units can be processed in parallel with loading a file into XEDIT by using the secondary output stream from *find*.

Creating a netdata file for transmission is in principle the reverse procedure. But now you need to be concerned with generating the control records with the proper format and contents. For a variable record format file, the record format declared in the transmission header should be X'0002', meaning variable records without descriptor words. Refer to INMR123 REXX S2 for an example of how to construct this header.

IEBCOPY Unloaded Data

There is a fair amount of work to do if you receive an IEBCOPY unloaded data set on tape or via RSCS from a TSO user. Data are blocked several ways. Looking “inside out”:

- IEBCOPY unloads a data set by prefixing 12 bytes of information (including the count field: FMBBCHHRKDD) to the key and data fields of the physical record (the block) it has read from the disk; these records are blocked within the logical record length of the output data set.
- The output data set is normally written in variable spanned (VS or VBS) format, which prefixes record and block descriptor words.
- If sent by the TRANSMIT TSO command, the netdata headers and trailers are added, and the complete data set is blocked in the netdata format described earlier.

The sample OSPDS REXX shows how to process the IEBCOPY unloaded data set after the physical blocking has been taken care of. The example creates a file for each member of the partitioned data set, or a stacked file where members are separated by *COPY delimiter records.

Building a Selection Key

When there is no built-in selection stage that performs the selection you require, you should consider prefixing the record with a search key, selecting the record using the new key, and finally discarding the key. For example, to select records in which the second word contains one or more of the characters #0\$ (assuming the word is ten characters or shorter):

⁷ The pipeline specification is split to fit within the column width.

Figure 138. Building a Search Key

```

| ...
| spec word 2 1 1-* 11',          /* Prefix second word      */
| xlate 1.10 00-ff blank # x @ x $ x',          /* Rub out */
| locate 1.10 /x/',
| spec 11-* 1',                  /* Remove key              */
| ...

```

The first *spec* stage puts the second word of the record into the first ten columns of the output record; the original input record is appended from column 11 and onward. The *xlate* stage works on the first ten columns only. It translates everything to a blank, except for the number sign, the at sign, and the dollar sign, which are translated to “x”. The *locate* stage then selects the records that contain an “x” anywhere within the first ten columns. The second *spec* stage extracts the original record, dropping the selection key.

While the previous example of *pipethink* is ingenious and the best way at the time of writing, the selection can be performed directly as of *CMS Pipelines 1.1.10*:

```

| ...
| locate word 2 anyof /#@$/',
| ...

```

Selecting, Revisited

Previously, you saw how to select records with the string “ PIPERM ”. This selects all such macro instructions when applied to an assemble file, but it might also select lines where the comment includes the word. To improve on this, first consolidate continuations and remove comment lines beginning with “*” or “.*”:

Figure 139. Advanced Selection

```
... | asmcont | nfind *| nfind .*| locate / PIPERM / |...
```

Adding a column range to *locate* finds exactly the lines you want if the program has all operation codes in column 10. The three commands in Figure 140 give identical results:

Figure 140. Finding Records with a Particular Operation Code

```
... | locate 9.8 / PIPERM / | ...
... | find' left(' ',8)'_PIPERM_| ...
... | zone 9.8 find _PIPERM_| ...
```

However, the operation codes do not have to begin in column 10. Figure 141 shows how to locate lines with a particular operation code, independent of its position in the input record.

Figure 141. More Advanced Selection

```
... | strip leading to blank | strip | find PIPERM_| ...
```

Here the label is removed by stripping to the first blank character. The next *strip* removes blank characters after the label (and at the end of the record); *find* selects lines with the operation code followed by at least one blank.

Using Many Filters

But you really want the first positional operand, so continue as follows:

Figure 142. Getting the Message Number

```
... | strip leading to blank | strip | chop anyof /, / | ...
```

Here the operation code is deleted in the same way that the label was removed above. The record is truncated before the first blank character or comma, which is the end of the first positional operand. Each line is now just a single word.

This may not seem overly useful. But when the file name and type are added by *spec*, *fanin* joins the lines from all programs, and the stream is sorted a few ways, then you have a handy cross reference. The complete set of selection stages is shown in Figure 143:

Figure 143. Complete Message Number Extract Pipeline

```
/* Extract Message Numbers */
'pipe',
'disk' fn 'assemble|',          /* read file          */
'asmcont|',                    /* expand continuation */
'nfind *|',                    /* No comments        */
'nfind .*|',                   /* No macro comments  */
'locate / PIPERM /|',         /* discard most        */
'strip leading to blank|',     /* Rid us of label     */
'strip|',
'find PIPERM_|',              /* Retain only opcodes */
'strip leading to blank|',     /* and discard...      */
'strip|',
'chop anyof /, /|',          /* retain number       */
'spec /'fn'/ 1 1-* 10|...
```

You may find this approach contorted, but no doubt you agree that a message cross reference can be useful. The question is, would you have written a program to do the same or would you have relied on a manual system?

Though many built-in programs have been added to *CMS Pipelines* since the section above was written in 1985, the usefulness of a cascade of filters is not in dispute; learning *pipethink* is just as important today as it was then. Still, using vintage 1994 technology, this selection might be performed more naturally by this cascade:

Figure 144. Another Complete Message Number Extract Pipeline

```
'PIPE (name PIPEUG)',
'|<' fn 'assemble',          /* Read file          */
'|unpack',                    /* If the files are packed */
'|asmcont',                    /* Join continuations    */
'|strnfind /*/',              /* Delete open code comments */
'|strnfind /*./',             /* Delete macro comments */
'|change 1 / /. /',           /* Ensure all lines have a label */
'|pick word 2 = /PIPERM/',     /* Look for the operation code */
'|spec word 3 1',              /* Get operand field      */
'|spec /'fn'/ 1',              /* Get file name          */
'|/?/ 10',                      /* Default of ?          */
'|fieldseparator ,',          /* Prepare to address operands */
'|field 1 10',                 /* Get first operand      */
'|...
```

The first *spec* stage puts the operand field into column 1 of the record. The second *spec* stage inserts the file name and extracts the message number in a rather subtle way. A question mark is inserted into column 10 in case there is no first positional operand; the field separator is set to a comma and the first field is extracted and inserted into the output record, overlaying the question mark. If, however, the first positional operand is omitted, the input record will contain a comma in column 1, the field will be null, and the question mark will be left unchanged.

You can easily create complex record validation functions. For example, to ensure that columns 1 to 10 contain only one word of numeric data without regard to the placement of the number within the ten columns:

Figure 145. Validating Numbers

```
... | verify 1-10 / 0123456789/ | nlocate substr word 2 of 1-10 | ...
```

The *verify* stage selects records that contain blanks and digits in the first ten columns, but this could still contain a string that is not a single number. Thus, the *nlocate* stage is employed to discard records that contain more than one word in the first ten columns.

This is still not perfect, for Figure 145 accepts a record that contains all blanks in the first ten columns. To be sure that there is a word you must add a *locate* stage:

Figure 146. Ensuring that a Number Is Present

```
| verify 1-10 / 0123456789/
| nlocate substr word 2 of 1-10
| locate substr word 1 of 1-10
```

To validate a number with a decimal point, simply tag on another stage to reject the third field, using a period as the field separator:

Getting File Information

Figure 147. Validating a Decimal Fractional Number

```
verify 1-10 / .0123456789/  
nlocate substr word 2 of 1-10  
locate substr word 1 of 1-10  
nlocate substr fieldsep . field 3 of 1-10
```

Obtaining Information about Files

CMS files

Use *command* LISTFILE to obtain file status information on files.

Figure 148. *command* LISTFILE Example

```
command LISTFILE * TEXT (NOH ALLOC | sort | console  
▶$$PIPE TEXT A1 F 80 4162 82  
▶EXTEPT TEXT A1 F 80 12 1  
▶PIPGDT TEXT A1 F 80 8 1  
▶PIPRUN TEXT A1 F 80 31 1  
▶TT TEXT A1 F 80 3 1  
▶Ready;
```

In Figure 148 the standard CMS command was run and the output intercepted. This works fine when few files are listed, but because the command has to complete before the lines can be processed by the pipeline, you may run out of storage if you wish to process all files on all accessed mode letters. You may be able to list files on one mode at a time to minimise buffer requirements.

Use *state* and *statew* to obtain information about specific files. Note in Figure 149 the similarity to the command drivers: an initial item can be specified with the stage; further input lines list more files you wish information about.

Figure 149. *state* Example

```
literal * text a|state * script *|  
▶DSMUTOC SCRIPT A1 V 102 48 1 11/24/86 19:35:  
▶TT TEXT A1 F 80 3 1 11/23/86 16:43:  
▶Ready;
```

Figure 150 on page 71 shows how the function of VMFDATE is implemented using *state*.

Figure 150. Variation on VMFDATE

```

/* Generate the information on a file */
/*      5785-RAC (RPQ P81059 5799-DKF) - CMS PIPELINES */
/*              (C) COPYRIGHT IBM CORP */
/*      LICENSED MATERIAL - PROGRAM PROPERTY OF IBM */
/* REFER TO COPYRIGHT INSTRUCTIONS FORM NO. G999-0001 OR G120-2083 */

/*****
/* Change activity: */
/*12 Sep 1993  +++ Rename. Make sensitive to system. */
*****/

signal on novalue

Parse source where . myfname .
parse arg file
parse var file fn ft fm .

compid=left(myfname, 3)
If where='CMS'
  Then signal notcms

'callpipe (end \ name PIPDATE.REXX:22)', /* Subroutine pipeline */
'|*:', /* Input files */
'|literal' file, /* Inject args */
'|s:state', /* Look for it */
'|spec 1-22 1 28.7 next 37-44 next 56-* next',
', /* Re-arrange */
'|spec 1-* 1.80 right', /* Right-adjust */
'|*:', /* Write to output */
'|s:', /* Files not found go here */
'|change ,,File not found: ,', /* Garnish it */
'|console' /* Say so */
exit RC

notcms:
'callpipe (name FPLDATE)',
'|listispf dd='ft fn,
'|stem f.'

If f.0=0
  Then exit 28

If f.1=''
  Then exit

'callpipe (name FPLDATE)',
'|literal' space(f.1),
'|spec 1-* 1.80 right',
'|*:'
exit RC

```

The command CALLPIPE is used in a REXX program to call a subroutine pipeline as described in “Using CALLPIPE to Run a Subroutine Pipeline” on page 103.

Getting File Information

TSO data sets

`state` determines whether a data set or an allocation exists. When the data set or allocation exists, the fully qualified data set name is written to the primary output.

Figure 151. Using `state` on TSO

```
pipe state names.text | console
▶DPJOHN.NAMES.TEXT
▶READY

pipe state dd=rexx | console
▶DPJOHN.TSO.TREXX
▶READY
```

You can use REXX functions or issue TSO commands to obtain other information about a data set:

Figure 152. Using TSO Commands to Obtain Information about a Data Set

```
pipe tso lista | take 3 | cons
▶DPJOHN.TSO.LOAD
▶SYS1.HELP
▶DPJOHN.PIPE.HELPLIB
▶READY

pipe tso listds names.text | cons
▶DPJOHN.NAMES.TEXT
▶--RECFM-LRECL-BLKSIZE-DSORG
▶ VB 255 8192 PS
▶--VOLUMES--
▶ TA922A
▶READY
```

Three device drivers provide information without issuing TSO commands. `listcat` provides data set names that share a qualifier or part of one; `listdsi` provides detailed information about individual data sets; and `sysdsn` tests for the presence of specified data sets.

To see which data sets you have catalogued that begin with the letter T:

Figure 153. Using `listcat` to Obtain Data Set Names

```
pipe listcat t | cons
▶PIPER.TFTP.FILTERS
▶READY
```

The pipeline was run while the prefix was set to `PIPER`. Note that you do not append an asterisk to the search criterion.

Figure 154. Using *listdsi* To Obtain Data Set Information

```

pipe listdsi tftp.filters | cons
▶=SYSDSNAME=PIPER.TFTP.FILTERS
▶=SYSVOLUME=RE9T01
▶=SYSUNIT=3390
▶=SYSDSORG=P0
▶=SYSRECFM=VB
▶=SYSRECL=255
▶=SYSBLKSIZE=8192
▶=SYSKEYLEN=0
▶=SYSALLOC=1
▶=SYSUSED=1
▶=SYSPRIMARY=1
▶=SYSSECONDS=1
▶=SYSUNITS=CYLINDER
▶=SYSEXTENTS=1
▶=SYSCREATE=1992/063
▶=SYSREFDATE=1994/298
▶=SYSEXDATE=0
▶=SYSPASSWORD=NONE
▶=SYSRACFA=GENERIC
▶=SYSUPDATED=NO
▶=SYSTRKSCYL=15
▶=SYSBLKSTRK=6
▶=SYSADIRBLK=
▶=SYSUDIRBLK=
▶=SYSTEMBERS=
▶=SYSREASON=0000
▶READY

```

Under the covers, *listdsi* uses the REXX function by the same name and then writes the variables set by REXX into the pipeline.

Figure 155. Using *sysdsn*

```

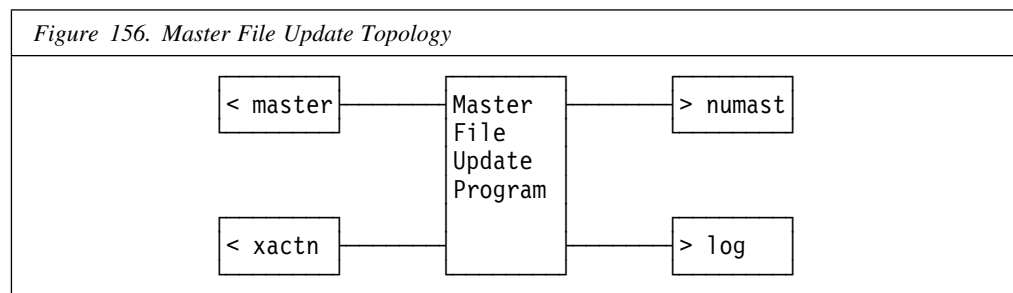
pipe sysdsn tftp.filters | console
▶OK
▶READY

```

sysdsn exposes the REXX function by the same name; the function result is written to the pipeline.

Chapter 5. Using Multistream Pipelines

In the previous chapters, we have seen pipelines of programs that read and write a single stream. In contrast, consider a program to update a master file. It needs to read two files, the master file and the transaction file. And it writes two files, an updated master file and a log file. Figure 156 shows the topology of a pipeline with a program that uses two input streams and two output streams.



CMS Pipelines is unique in supporting a multistream pipeline topology. In such a topology, a stage can have any number of input and output streams. Because there was no established paradigm for expressing such a topology, one was invented. The current implementation evolved after several experiments.

An implementation of multistream pipelines consists of a programming interface to allow a program access to the streams as well as rules for specifying the topology in the PIPE command. We shall deal with the programming interface in Chapter 7, “Writing a REXX Program to Run in a Pipeline” on page 97; this chapter is concerned with built-in programs that support multiple streams.

Though it has not been mentioned explicitly yet, the pipeline specification “works” by specifying a sequence of transformations and functions that are applied to the data as they flow from left to right through pipelines such as those presented this far in the book. In the simple pipelines, putting the definitions of two stages next to each other with a stage separator between them has been sufficient to define the input/output relations. The output stream of one stage is connected to the input stream of the following one. That is all that is necessary to define the connection on the *primary stream*.

Streams are defined in pairs; a stage always has a primary input stream and a primary output stream. When the stage is first in a pipeline, the primary input stream is *not connected*, but it is there all the same. Likewise a stage that is last in a pipeline has an output stream, even though it is not connected.

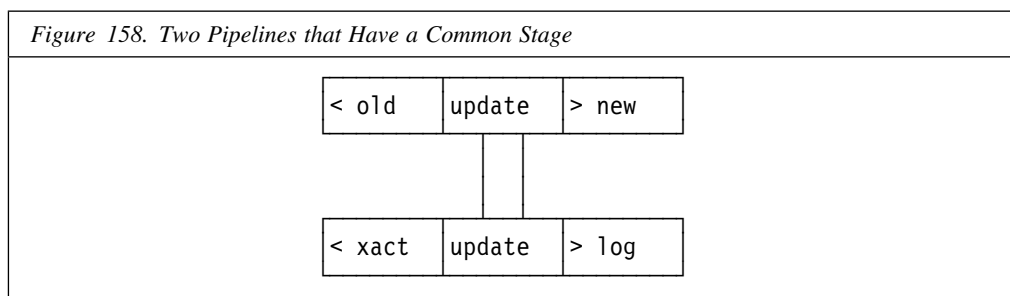
The generalisation to multiple streams in a stage is just that the stage can have multiple streams; the difficult part is how to specify the topology. *CMS Pipelines* has taken the model that a multistream pipeline consists of two or more individual pipelines, each of which is specified just like a simple pipeline is. The trick is that a multistream stage is in more than one pipeline.

To express multiple pipelines in a single command string, an *end character* is used to separate pipelines, just as a stage separator separates stages. There is no default end character; it is specified in each multistream pipeline and it can be different in each. All will be explained in “Options” on page 237; suffice it to say here that to use the question mark as an end character, you can write the beginning of the pipeline like this:

Figure 157. A Pipeline Specification Having Two Pipelines

```
pipe (endchar ?) <pipeline-1> ? <pipeline-2>
```

ENDCHAR can be abbreviated as END; and it usually is. Refer back to Figure 156 on page 74. It looks very much like two pipelines, one for the master file and one for the transactions, does it not? If the update program reads and writes the master file on the primary stream (*update* does that), clearly the device drivers to read and write these files should be connected to the primary streams for update; this is very much like a simple pipeline. Likewise, the second pipeline would read the transactions into the update program's secondary stream and write its output records to the log file.



This might be written as in Figure 159.

The pipeline in Figure 159 is written as a *left-handed pipeline*; that is, lining the stage separators and the end characters on the left. As far as REXX is concerned, it is a character string all the same. The pipeline has also adopted the convention of prefixing the first pipeline with the end character. Note that the left-handed style makes for easy recognition of the pipeline end character.

Figure 159. Almost an Update Pipeline

```
'PIPE (end ? name PIPUMSP)',
  '?< master file',           /* Read master          */
  '|update',                 /* Perform update       */
  '|> new master a',        /* Write new master     */
  '?< xaction file',        /* Read transaction file */
  '|update',                 /* Apply the transaction file ?? */
  '|> log file a'           /* Write log file       */
```

The pipeline specification above almost does the update, but there is one serious flaw in it: the update program is invoked twice with one set of streams on each invocation, rather than once with two sets of streams. We need to specify a connection between the two stage positions where the update program connects to its neighbours. This is done with a label, which is a string of one to eight characters followed by a colon. This example uses a label that is one character:

Multistream Pipelines

Figure 160. An Update Pipeline

```
'PIPE (end ? name PIPUMSP)',
  '? < master file',          /* Read master      */
  '|u: update',              /* Perform update   */
  '| > new master a',        /* Write new master  */
  '? < xaction file',        /* Read transaction  */
  '|u:',                     /* Apply transaction */
  '| > log file a'           /* Write log file    */
```

All occurrences of a label in a pipeline specification (the formal name for the argument to the PIPE command) refer to a single stage. The stage has as many pairs of input and output streams as there are occurrences of the label. The actual program to run and its arguments are specified the first time the label is used. This is called the definition of the label. The next time the label occurs is a reference back to the previous definition. Because the stage has already been specified completely, the label is written by itself at the point in the pipeline topology where the stream should be connected. The first reference to the label creates the secondary input and output stream; the next one creates the tertiary streams; and so on.

Note in Figure 160 that both the label definition and the label reference are in the middle of a pipeline; each connects an input stream as well as an output stream. The most common beginner's error is to specify two label references where only one should have been used.

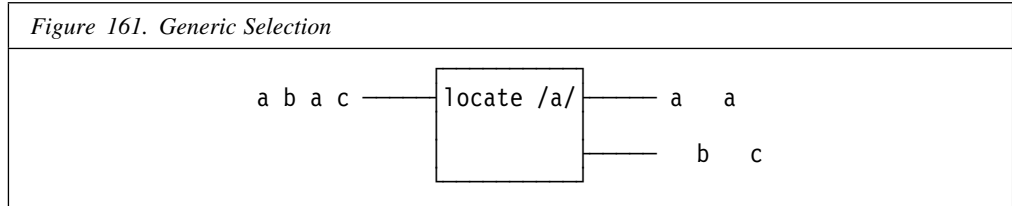
In summary: A stage can be in two or more pipelines at the same time and have access to two or more data streams, one in each pipeline. Two concepts are introduced to support this:

- A stage can be referenced several places in a set of pipelines if its definition is prefixed with a label (up to eight characters followed by a colon). The first time a label is used is the *primary stream* (number 0) for that stage. The *secondary stream* is defined the second time the label appears, and so on.
- To be able to use more than one device driver that must be first in a pipeline, the character defined by the option ENDCHAR is used to delimit multiple pipelines in a single command string. (ENDCHAR is usually abbreviated to END).

Building Blocks for Multistream Pipelines

Though the master file update problem was the motivation to support multistream pipeline topologies and though *update* certainly supports two sets of streams, this is no longer considered the mainstream. It soon became clear that multistream pipelines could be used for much more.

locate was the first selection stage to be modified to write rejected records to its secondary output stream; this opened the field of transformations that depend on record contents, because records could be processed differently when they were selected than when they were rejected.



In Figure 161 the individual characters represent records. The relative sequence is shown by their order from left to right. *locate* passes the records that contain “a” to its primary output stream and the records that do not contain “a” to its secondary output stream.

But how do we merge these records back into one file after they have been processed? *faninany* does precisely this:

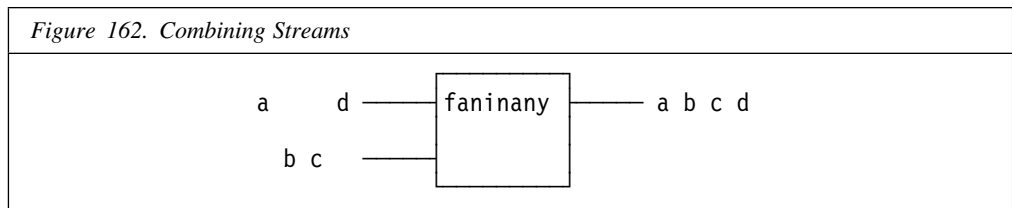


Figure 163 shows how to combine the two ideas, to prefix selected lines of a file with a marker. The first part of the figure is the EXEC that was run (the command is too long to type directly). It is followed by the response and a topology diagram.

Figure 163. Sample Multistream locate

```

/* Sample multiline locate                                     */
signal on novalue
'PIPE (end ? name SAMPLOC)',                               /* Declare ? as end-character */
'| < small exec',                                         /* Read input file           */
'|a: locate /y/',                                          /* Look for y                 */
'| change //---> /',                                       /* y's get adorned here     */
'|b: faninany',                                           /* Merge with non-y's       */
'| console',                                              /* Write to terminal         */
'?a:',                                                    /* non-y's come here       */
'| change //      /',                                       /* Shift them to col 5     */
'|b:'                                                      /* Merge with y's          */

```

```

samploc
▶ /* This is a small Exec */
▶ signal on novalue
▶---> say 'Bye...'
▶Ready;

```

Combining Data Streams

If you want to combine several data streams, the easiest way is to run the pipelines one after another and accumulate the output by appending to a file:

Figure 164. Simplistic COPYFILE Append

```
/* simplistic copyfile append for CMS */
arg fn1 ft1 fm1 fn2 ft2 fm2 fn3 ft3 fm3 .
'pipe <' fn1 ft1 fm1 '|' >' fn3 ft3 fm3
r=RC
'pipe <' fn2 ft2 fm2 '|' >>' fn3 ft3 fm3
exit max(r, RC)
```

```
/* simplistic copyfile append for TSO */
Address Attach
arg fn1 fn2 fn3 .
'pipe <' fn1 '|' >' fn3
r=RC
'pipe <' fn2 '|' >>' fn3
exit max(r, RC)
```

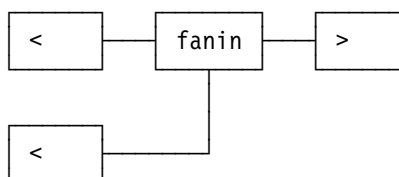
This is all proper, but with variable format files on CMS, performance of the second pipeline can be inferior to the first one. More importantly, there are cases where you wish to block the composite data stream and put it to a medium where you cannot write records piecemeal. Think of a tape file, for instance; you may not be able to obtain the result desired simply by running more than one pipeline, because this might create a short block in the middle of the output file.

The gateway *fanin* combines the input on all its input streams, writing the data in the order specified. Output is a single data stream:

Figure 165. Not So Simplistic Append

```
/* Not so simplistic append for CMS */
arg fn1 ft1 fm1 fn2 ft2 fm2 fn3 ft3 fm3 .
address command
'ERASE' fn3 ft3 fm3
'PIPE (end ?)<' fn1 ft1 fm1 '|a: fanin|>' fn3 ft3 fm3,
      '?<' fn2 ft2 fm2 '|a:'
exit RC
```

```
/* Not so simplistic append for TSO */
arg fn1 fn2 fn3 .
address Attach
'PIPE (end ?)<' fn1 '|a: fanin|>' fn3,
      '?<' fn2 '|a:'
exit RC
```



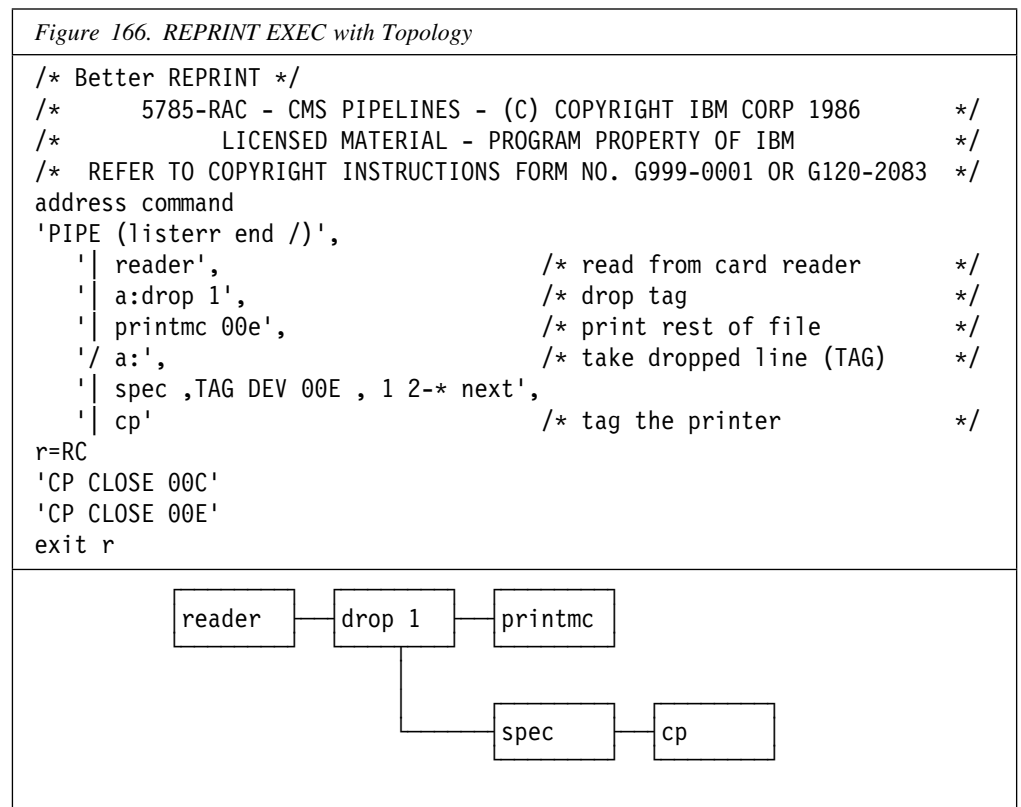
Here you see two pipelines defined. The option ENDCHAR (abbreviated to END) specifies that the question mark (?) separates pipelines. The end character can be any character (that has no other special meaning to the scanner), but it cannot be used in the individual pipelines unless it is escaped.

The intersection occurs in the *fanin* stage, which has the label *a:*. The label is *defined* on the first line of the pipeline specification; it is *referenced* on the next line. The primary pipeline is selected when *fanin* starts. It passes all input records to the primary output stream; when *fanin* reaches end-of-file on the primary input stream, it switches to the secondary input stream and then passes those records to the primary output stream.

The example in Figure 165 on page 78 shows how to use multiple streams in general. This particular example can also be done with *append* as was shown in Figure 71 on page 38.

Splitting a Data Stream

Figure 166 shows how to create a copy of a printer SPOOL file which resides in your virtual reader:



The file is read from the reader and split into two streams by *drop*, which removes the tag record from the primary input stream and sends it to the secondary output stream. The primary stream carries the file itself; it is sent to the printer. The secondary stream extracts the tag from the first record of the file (discarding the CCW operation code in the first position) and sets the tag for device 00E using *cp*. This works because the tag in the SPOOL file is set when the CP command “close” is issued after the pipeline has completed processing.

Generating a CMS Macro Library

maclib generates a macro library from a file where members are separated by records containing *COPY in column one and the member name after a blank. Being a filter, it is concerned only with what is inside the library, not how it is written to disk⁸. One difficulty is that the first record of a library must point to the directory, but the directory is after the body of the file: its position cannot be computed until end-of-file. To buffer the complete file inside *maclib* might cause storage overruns. Thus, a different approach was adopted using three output streams:

- 0 On the primary output stream from *maclib* comes a dummy 80-byte record as a placeholder. This is followed by the members of the library, each having an end of member record. *COPY records are discarded.
- 1 The directory is written to the secondary stream as each record is complete. This is buffered until the library is complete, appended to the contents of the library (stream 0), and written to disk.
- 2 The dummy first record of the library indicates that the library has no members. The real first record of the library is the last record written; it has a data stream to itself. This stream is connected to the secondary input stream of the > stage that writes the library and the index. > replaces the dummy pointer record with the real one.

Figure 167. Pipeline Topology while Generating a Macro Library

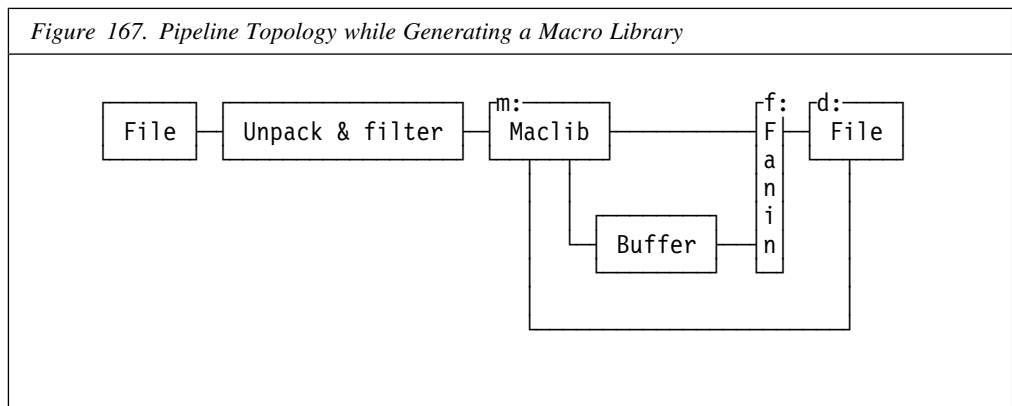


Figure 168. GENMLIB EXEC

```

/* Generate a macro library                                */
signal on novalue
arg fn ft fm .
address command
'PIPE',
  '<' fn word(ft 'copy', 1) fm, /* read the file      */
  '| unpackx', /* Unpack if packed */
  '| macwrite' fn /* Write MACLIB    */
exit RC

```

Figure 168 shows GENMLIB EXEC, which generates a macro library from a packed COPY file. The first two lines of the pipeline specification read the file and unpack it (if it needs to be unpacked). The last stage names the REXX program in Figure 169 on page 81.

⁸ *maclib* also creates a TXTLIB, acceptable to *members*, from TEXT decks, as long as there is a *COPY record between members.

Figure 169. MACWRITE REXX Subroutine Pipeline

```

/* MACWRITE REXX: Does the hard part of MACLIB GEN          */
signal on novalue
arg fn ft fm recfm lrecl .
If fn=''
  Then exit 999
If lrecl=''
  Then parse value arg(1) subword('maclib a fixed 80', ,
    words(arg(1))) with fn ft fm recfm lrecl
plrecl=lrecl
If abbrev('VARIABLE', translate(recfm))
  Then lrecl=''
'callpipe (end ?)',
  '?*:',
  '|pad' plrecl,
  '|m:maclib',
  '|f:fanin',
  '|d:>' fn ft fm recfm lrecl,
  '?m:',
  '|o:fanout',
  '|buffer',
  '|f:',
  '?m:',
  '|d:',
  '?o:',
  '|fblock 16',
  '|chop 8',
  '|nfind' '00'x,
  '|sort count',
  '|nlocate 1.10 , 1,',
  '|spec /Duplicate member in' fn': / 1 11-* next 1.10 next',
  '|console'
/* Subroutine
/* Get input
/* Pad to length
/* Generate MACLIB
/* Add directory
/* Write to disk
/* The directory
/* Process directory twice
/* Buffer it
/* To write
/* The first record
/* Replace first record
/* Directory records
/* Get each entry
/* Just the name
/* Drop deleted/padding
/* Sort'm
/* Discard singles
*/
exit RC

```

Figure 169 shows a subroutine pipeline to generate a macro library. Subroutine pipelines are described in Chapter 7, “Writing a REXX Program to Run in a Pipeline” on page 97. The construct “*:” at the beginning of the subroutine pipeline means that it should be connected to the current input of the stage that issued the CALLPIPE pipeline command.

The *maclib* stage has one input stream (its primary) and three output streams. The primary output stream is connected to the primary input stream of >. The secondary output stream is connected to a *buffer* stage where it is accumulated. The tertiary output stream is connected to the secondary input stream of >.

Figure 167 on page 80 shows a diagram of the topology of the part of the subroutine that creates the library. The last pipeline (which begins '?o:',) is not shown; it processes a copy of the directory (made with *fanout*) to determine whether the file has duplicate member names. It writes a diagnostic to the terminal for each name that occurs more than once.

GENMLIB also accepts variable record format SCRIPT files that have *COPY records to identify members.

Multistream Pipelines

If you wish to process the contents of the newly generated macro library without writing the file to disk, you must use *buffer* to delay the main part of the file until the correct first record is written to the secondary output stream. Figure 170 on page 82 shows how to create a single stream with the contents of a macro library. It relies on the ability to specify the order *fanin* is to process its inputs.

Figure 170. Sample MACLIB Stream

```

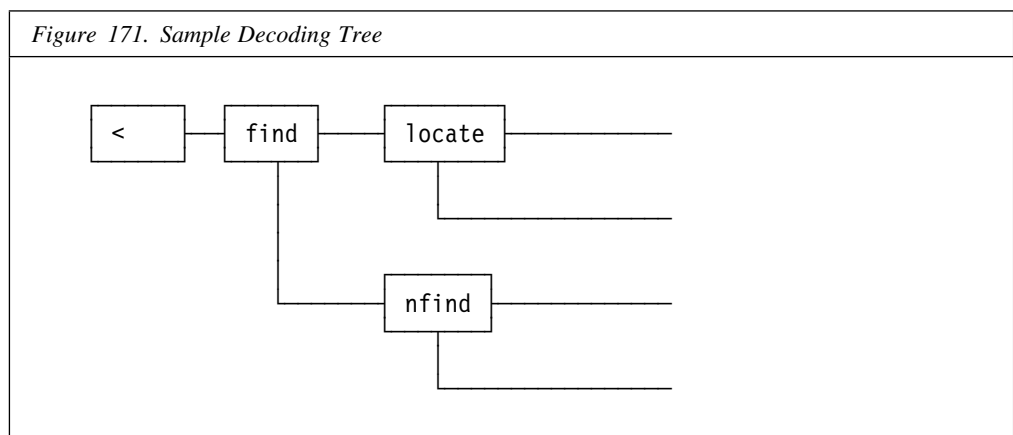
!      /* Generate a MACLIB stream                                     */
!      'pipe (end ?)',
!      .....
!      '| mac:maclib',
!      '| drop 1',
!      '| buffer',
!      '| gthr: fanin 2 0 1',
!      ..... Here is the maclib as a stream
!      '? mac:',
!      '| buffer',
!      '| gthr:'
!      '? mac:'
!      '| gthr:'

```

Decoding Trees

Selection stages emit data on one of two pipelines. They can be cascaded to any depth. Create multiple identical streams with the gateway *fanout*. (And you can write your own in REXX if these do not meet your needs.) You now have a way to process the file with particular filters, depending on the contents of the record. Here is a decoding tree:

Figure 171. Sample Decoding Tree



An input record appears on exactly one output line as long as the decoding tree consists exclusively of selection stages; records can be processed differently on each stream out of the decoding tree. Though you can write a file at the end of each stream, you can also gather it all into a single stream again. Use the gateway *faninany* to gather the records as they appear on the various lines. If you want a different ordering, you must buffer records you wish to defer. *buffer* and *sort* do that; then use *fanin* as already shown.

Remembering Past Data

Consider a file that describes users. For each user there is a group of records in this format:

Figure 172. Not so Hypothetical File Format

```
User <userid> <privileges>
Link <resource> <access mode>
:
Link <resource> <access mode>
```

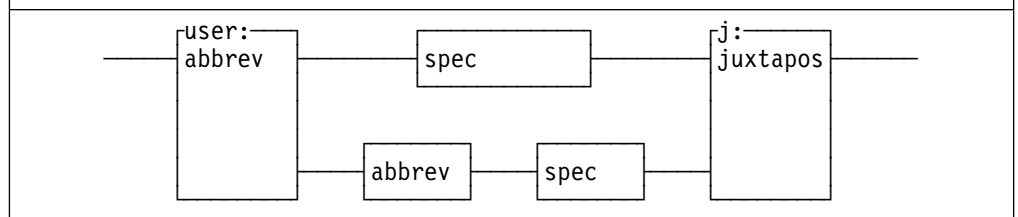
The first word of the user record contains the word “user”, which can be abbreviated to one letter. Likewise the first word of the resource record contains “link”, also with a potential abbreviation to one letter. To allow for pretty formatting, leading blanks are allowed in this file; case is ignored.

The question is, how to construct a file that lists which users have access to what resources. You could write a REXX filter to perform this function, but since writing filters in the REXX programming language is described in a later chapter, you must resort to *pipethink* to develop a solution consists entirely of built-in programs.

The solution seems to hinge on selecting records based on abbreviations and how to remember the user ID that applies to a resource record.

A quick check in the help menu shows *abbrev* conveniently near the upper left corner; it seems to do the job. As for remembering the user ID, the built-in program is *juxtapose* (which means putting things next to each other). The topology of the solution (after the initial strip and capitalisation) is:

Figure 173. Topology of One Record Storage



The first (the larger) *abbrev* stage selects user records, which travel on the upper pipeline where the second word is extracted and delivered to *juxtapose*'s primary input stream padded out to ten characters. *juxtapose* stores the contents of the record in a buffer and discards the record.

Resource records travel on the lower pipeline, where the second and third words are extracted and passed to the secondary input stream of *juxtapose*.

When *juxtapose* senses a record on its secondary input stream, it appends this record to the one stored in its buffer and writes the composite record to its primary output stream. Thus, the user ID has been prefixed to the resource name.

Multistream Pipelines

Figure 174. Prefixing User ID to Resource Name

<pre> /* RESBYU EXEC -- Get resource ID by user name. */ Signal on novalue Address COMMAND 'PIPE (end ? name RESBYU)', '? < resource file', /* Read input file */ ' strip', /* Remove leading/trailing blanks */ ' xlate', /* Make uppercase */ ' user: abbrev USER 1', /* Select user records */ ' spec word 2 1.10', /* Extract user ID */ ' j: juxtapose', /* Prepend to resource */ ' console', /* Display result */ '?user:', /* Non-user records come here */ ' abbrev LINK 1', /* Select resource records */ ' spec word 2 1.8', /* Extract user ID */ ' word 3 nextword', /* Extract resource name */ ' j:' /* Pass it to be prefixed with the user ID. */ exit RC </pre>	
<pre> * Resource File for system Z User john paswd g Link maint 190 190 rr Li preben 191 391 rr U preben xpvt abcdegh Link maint 190 190 rr L john 191 391 rr </pre>	<pre> resbyu ▶JOHN MAINT 190 ▶JOHN PREBEN 191 ▶PREBEN MAINT 190 ▶PREBEN JOHN 191 ▶Ready; </pre>

Note: In this example, the input records to *juxtapose* were derived from the same data stream and the filters in the multistream path did not allow records to get out of order; that is, they do not “delay the record”. (This concept is described later.)

juxtapose is likely to give unexpected results when the two input streams are derived from sources, such as independent device drivers, that can produce records concurrently. When records are available at both input streams at the same time, it is unspecified and unpredictable in which order the streams will be read. Given a chance, *juxtapose* might drain all records from its primary input stream before reading any records from the secondary input stream.

You can use *synchronise* to make records on two streams march in lockstep, but it is probably easier to use *spec* or *overlay* to combine sets of input records into one output record.

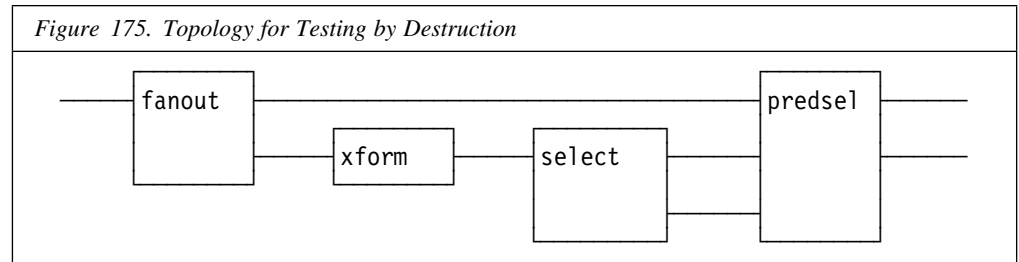
Destructive Testing

Examples in earlier chapters have shown how to prefix a key field to records, perform some processing, and delete the key field afterwards. This, for example, allows for sorting or selection based on a transformation of the record or parts of the record. To be able to delete the key field, it must have a fixed length or be terminated by some particular character, or in other ways be identifiable.

When adding a key field cannot bend a selection stage to your needs, you may consider the technique of testing by destruction. The crux of this technique is to keep a copy of the original record in a safe place while the the record to be tested is transformed in an irre-

versible way and then passed to the selection stage. But the output from the selection stage is not used as output data; rather, it is used to control which output stream is to receive the original record.

This is a picture of the topology of such a setup:



In Figure 175, *fanout* and *predselect* represent the built-in programs, whereas *xform* and *select* are selected by you to perform the particular transformations and selection required. *xform* may represent a cascade of filters.

For each input record,

1. *fanout* writes the input record to its primary output stream, which is connected to the primary input stream of *predselect*.
2. *predselect* stores the record in a buffer for later use. (This is the safe place.) *predselect* then consumes the record without producing output.
3. *fanout* writes the same input record to its secondary output stream, which is connected to your suite of transformation stages.
4. The transformation stages mangle the record in whatever way they see fit and pass some concoction on to the primary input stream of the selection stage.
5. The selection stage selects or rejects. That is, it passes the input record to its primary output stream or the secondary output stream, depending on whichever criterion you have selected.
6. *predselect* detects an input record on its secondary input stream or its tertiary input stream. This record acts like a trigger.
7. *predselect* writes the contents of its buffer to the primary output stream if the trigger record was detected on the secondary input stream; it writes the contents of the buffer to the secondary output stream if the tertiary input stream was triggered.
8. *predselect* discards the trigger record.

Thus, if the transformation and the selection do not delay the record, the pipeline segment above as a whole will behave like a selection stage.

But there is one little extremely important detail that you must remember if you build your own selection stages using this approach: Do specify the option `STOP ANYEOF` on *fanout*; this makes sure that end-of-file can propagate backwards, both from the stages that follow in the pipeline and from the selection stage. If you make this kind of pipeline, it might be a good idea to test it with *tolabel* as the selection stage.

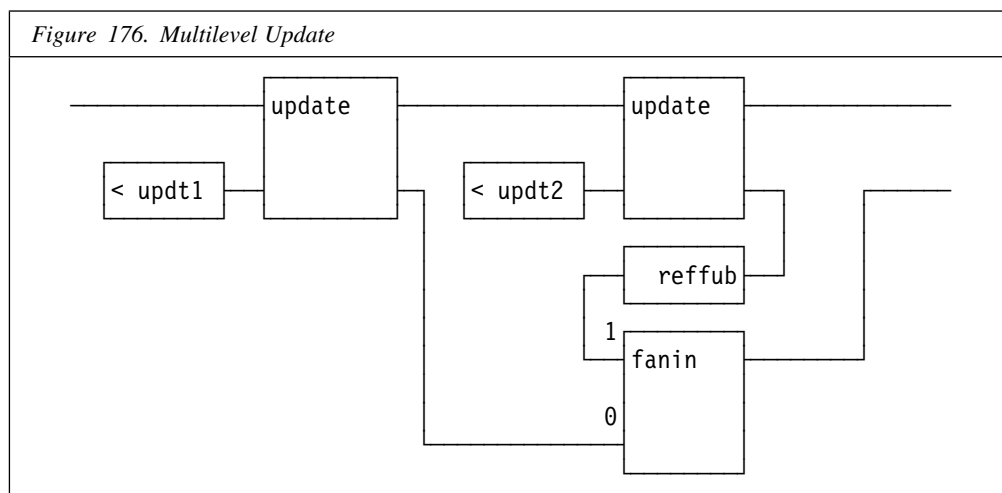
This technique is used under the covers of *casei* and *zone*.

Other Multistream Programs

CMS Pipelines provides many more multistream programs than the ones highlighted here. Refer to the entries for “multistream” in Appendix A, “Summary of Built-in Programs” on page 876 to see them all.

Update

update applies a single update to a data stream. The master file is on the primary pipeline; the update records are on the secondary one. Multilevel update is implemented as a cascade of *update* stages.



In this two level update, the master file travels on the top pipeline. The individual update files are read into the secondary input stream of the *update* stages that apply the updates. Unlike the CMS UPDATE command, the *update* stages apply the updates in parallel as the master file travels along the top pipeline.

This means that the log files are created in parallel too. But you probably do not wish to see the composite log file in the order that individual lines are written; presumably you would like (or more likely insist) that the first *update* stage’s log is first in the log file, followed by the second one, and so on.

To generate a file that aggregates files that are generated in parallel, all except the first file must be buffered in a *buffer* stage. The individual files are then fed to *fanin*, which will copy each file from its input streams as a whole to its primary output stream. In the figure we have taken the artistic licence to show the *buffer* stage backwards; records flow from right to left through it. We have also taken the liberty to show the secondary input stream to *fanin* at the top.

Note that all built-in programs are reentrant. It is perfectly proper to use a program several times; each invocation has (in general, at least) no information about other invocations; and it has no inclination to find out.

Multilevel update pipelines are usually generated by a program that reads the control file and the auxiliary control files to determine which updates are to be applied and then generate the appropriate pipeline. (Yes, you can even write a pipeline to do that.) Still, you might be curious how the topology in Figure 176 is coded in an EXEC. This is how:

Figure 177. A Two Level Update

```

'PIPE (end ? name PIPUMSP.SCRIPT:1018)',
  '?< program assemble',          /* Read base file          */
  '|u1: update first',            /* Apply first update      */
  '|u2: update last',            /* Apply last update       */
  '|> $program assemble a fixed 80', /* Write temporary assemble */
  '?< program updt1',             /* Read first update       */
  '|u1:',                          /* Apply it                 */
  '|log: fanin',                  /* Append second log file  */
  '|> program updtlog a',         /* Write composite log     */
  '?< program updt2',             /* Read second update      */
  '|u2:',                          /* Apply it                 */
  '|buffer',                       /* Wait for first one to complete */
  '|log:'                          /* Now append to log       */

```

Merge

merge reads records from all its input streams and writes the merged file to the primary output stream. When each input file is already sorted, the output file will also be sorted.

When *merge* sees records with the same key on two or more input streams, it writes the record in the order of increasing stream numbers. *merge* supports up to ten input streams. You could cascade *merge* stages, but you might run out of storage if this means reading a large number of disk files concurrently.

Collate

collate merges detail records into a master file. By default, the detail records follow the master record on the primary output stream. Master records for which there are no detail records are written to the secondary output stream. Unmatched detail records are written to the tertiary output stream. Additional plumbing is required to insert master records for which there are no detail records in the primary output stream; refer to the description in the reference part of this book.

In contrast to *merge*, *collate* supports only two input streams; it assumes that there is only one master record that has a particular key; and it allows you to specify whether the detail records should precede or follow the master record.

Lookup

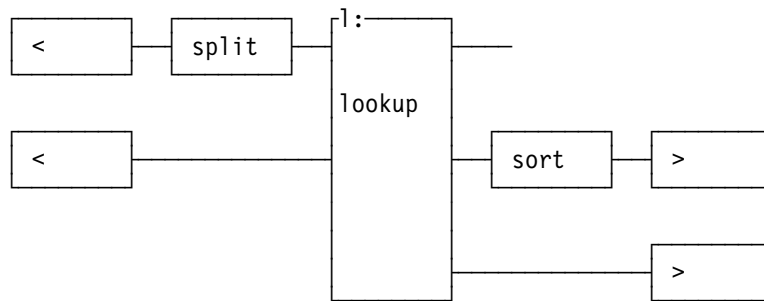
Use *lookup* to select records whose key field is one of several (or is not one of several). The secondary input stream to *lookup* contains the reference records. These records are read into *lookup* when it starts and are stored in a buffer internally while the file on the primary input stream is processed. When a record from the primary input stream has the same key as one of the reference records, the input record and the matching reference record are written to the primary output stream. Unmatched records from the primary input stream are written to the secondary output stream. When *lookup* gets end-of-file on its primary input stream, the reference records that were never matched are written to the tertiary output stream.

lookup is often used to remove “stop words” (words that are too common to be of interest) from a set of words being indexed:

Multistream Pipelines

Figure 178. Using lookup

```
/* Now get usage counts after stop words are discarded */
'PIPE (end ?)',
'|<' input_file,          /* Read input file          */
'|split',                /* Split into words         */
'|locate 2',             /* Ignore one-character words */
'|l: lookup',            /* Look up, discard matching */
'|< stop words',         /* Read file of stop words   */
'|l:',                  /* To lookup                */
'|sort count',           /* Get frequency counts     */
'|> word freq a',        /* Write result              */
'|?l:',
'|> unused stops a'      /* Stop words not present   */
```



lookup also supports dynamic change to the reference while it processes records from the primary input stream. This is useful when it is used in a server to validate the clients' authorities against a database that is updated dynamically. A record on the tertiary input stream is added to the reference, in effect adding a user or a privilege to the table; a record on input stream 3 causes the corresponding key to be deleted from the reference, in effect removing privileges.

Some Fine Points

You are justified in having a nagging suspicion that there is more to multistream pipelines than you have seen so far. And the exposition has indeed been simplified. In general, you should consider these issues when you write multistream pipelines:

- Make sure that data always can flow. If not, the pipeline will stall.
- Make sure that output records are in the order you expect. If not, you might get incorrect results.
- Make sure that end-of-file can travel backwards in your pipelines. If not, you might be spending system resources on something that produces no result.

The following sections contain a synopsis; the full story is in Chapter 22, "Scanning a Pipeline Specification and Running Pipeline Programs" on page 245.

Ensure the Pipeline Does not Stall

A pipeline is *stalled* when records cannot flow between stages. A stalled pipeline is as useful as a blocked drain.

Often a stall occurs because a stage that accesses multiple streams is trying to read from a stream on which no record will become available or is trying to write to a stream that will

not be read. To stall, a pipeline must also have more than one path between some stages. In general, you must consider the possibility of the pipeline stalling when there are one or more meshes in the pipeline topology; that is, when the pipeline contains two stages between which there is more than one path.

The *buffer* stage in Figure 167 on page 80 ensures that the inner loop in the pipeline does not stall; the fact that *maclib* severs its primary output stream and its secondary output stream before writing the final record to the tertiary output stream ensures that the outer loop does not stall either.

If a selection stage puts a fork in the pipeline, it is normal to join the two branches with *faninany*. Because *faninany* is able to read from any input stream that has a record available, it will be able to keep records flowing out of the multistream segment.

When records from a stage can travel over two or more paths to a *fanin* stage, you must ensure that records arrive at the inputs of *fanin* in the correct order, which is all records on the primary input stream, then all records on the secondary input stream, and so on. Buffer stages (*buffer* or *sort*) or *elastic* stages connected to all input streams other than the first will prevent stalls, at the expense of storage.

Keep the Order of Records

As you would expect (and no doubt also demand), the output records in Figure 163 on page 77 are in the same order as the input records.

This section describes how you can ensure the order of the output records from a decoding network that has been joined into one stream with *faninany*.

Two concepts are important to reason about the relative ordering of records:

- *locate* (and indeed every selection stage) passes the input record unmodified to an output stream. When both output streams are connected, a selection stage writes an input record once, and only to one of the output streams.
- *locate* (and indeed every selection stage), *spec*, and *faninany* are examples of built-in programs that do not *delay the record*. This means that output records are in the same sequence as the corresponding input record, and that an output record is produced before the corresponding input record is consumed. If the stages in the mesh do not delay the record, the output from *faninany* will be in the same order as the input to the selection stage.

Thus, in Figure 163 on page 77, *locate* does not read another input record until *console* has written the previous record to the terminal.

In general, to ensure that output records remain in the order they enter a pipeline segment having parallel paths, any part of the pipeline specification that can pass a record on parallel paths must consist entirely of stages that do not delay the record.

Some built-in programs (notably *specs*) synchronise their input streams; that is, they ensure that there is a record available on all their inputs before beginning a processing cycle. Connecting the outputs from *fanout* or *chop* to a synchronising stage without delaying the record will result in a stall. To ensure that the resulting output record is not delayed, you should insert a delay of one record into the topmost of the parallel paths between the *chop* and the *spec* stage. For example, this pipeline fragment reverses the first word of each line and puts it at the end of the line:

Multistream Pipelines

Figure 179. Reverse First Word on Each Line

```
/* Reverse first word of each line */
Signal on novalue
'callpipe (end ? name REWORD1.REXX:4)',
  '?*:',
  '|c: chop blank', /* Get label or null */
  '| reverse', /* Turn it round */
  '| copy', /* One-record delay */
  '|s: spec select 1 1-* next', /* Merge back */
  '| select 0 1-* nextword', /* At the end */
  '|*:', /* Pass on */
  '|?c:', /* Rest of record */
  '|s:'
exit RC
```

```
pipe literal abc def ghi | revword1 | console
► def ghi cba
►Ready;
```

Had the words in the output record been in the same order as in the input record, you could have used a cascade of *faninany* and *join* to perform the operation that is done with *specs* above.

Allow End-of-file to Travel Backwards

When you write REXX filters, be sure to test for the return code when writing output records, as well as when you read. When the program has only one output stream, there is no point in continuing after that stream is at end-of-file. You should terminate the program without consuming the input record that you are processing at the time you discover that you cannot write.

When you wish to trace data flowing in the pipeline to debug it, you should be careful not to insert a device driver directly in the main pipeline path, for a device driver does not propagate end-of-file backwards. After all, writing the output file is quite productive and there is no need to terminate just because further processing has terminated.

You can use *fanouttwo* to obtain a copy of the records that flow in the pipeline and be sure that end-of-file is propagated backwards and that the secondary output stream does not interfere with the main stream.

Figure 180. Safe Trace Capture

```
/* Capture a trace to a file */
parse arg file
'callpipe (end ?)',
  '?*:', /* Input records */
  '|x: fanoutwo', /* Pass on */
  '|*:', /* To main output */
  '|?x:', /* Copy if consumed */
  '|>' file /* Store it */
```

If your *CMS Pipelines* does not have *fanouttwo* you can approximate its behaviour with *fanout* STOP ANYEOF. This will not isolate the main pipeline from the device driver; if the device driver terminates for some reason, the main pipeline will also shut down.

Chapter 6. Processing Structured Data

The data layout of a file is often defined as a record in some programming language or by some other symbolic means.

Perform these steps to reference data in such a record symbolically:

1. Define the structure, for example as a file, but a conversion from some programming language definition is also possible.
2. Activate the structure definition by passing the structure definition to *structure* ADD.
3. Reference fields by using qualifiers and members in an *inputRange* or in a *spec* output specification.
4. Deactivate the structure definition when it is no longer needed by passing its name to *struct* DELETE.

Defining Structured Data

To start, consider the output from the CMS command QUERY ACCESSED:

```
pipe cms q accessed | take 5 | console
▶Mode Stat      Files Vdev  Label/Directory
▶A    R/W        156 191  SRV191
▶C    R/O        198 DIR   SFS:RVDHEIJ.PUBLIC
▶D    R/W        120 DIR   SFS:JOHN.REGTEST
▶E    R/W        973 DIR   SFS:JOHN.PIPELINE.TESTS
▶Ready;
```

You can reference each field by its column number, for example, the virtual device number begins in column 23, but perhaps it would be clearer to refer to it simply as Vdev.

Someone would have to make this definition manually, as the format is not present in a machine readable form. One possible definition is shown in Figure 181 below. Assume it is stored in the file QACC RECORD. We show one structure in this example, but such a file can contain multiple structure definitions.

Figure 181. Sample Structure Definition

```
: qacc
  Mode Length 1
  -    Length 1
  Extn Length 1
  -    Length 4
  -    Length 2
  Stat Length 1
  Files Length 10
  -    Length 2
  Vdev Length 4
  -    Length 2
  Label Length *
  Labelc(*) member Label Length 1
```

The file is in free format. You can span the definition across as many or as few lines as you like.

Structured Data

A structure is defined by a colon, which is followed by the name of the structure (blanks are optional after the colon). The fields are then defined on subsequent lines in this example.

Structure names and field names are case sensitive unless the structure is defined as caseless, that is, `STRUCT ADD ANYCASE` was specified to define it. They must begin with a letter from the English alphabet or one of the characters “@#\$!?”. Subsequent characters may also include the digits 0 through 9. This is the same syntax as a valid REXX simple variable, but unlike REXX, it can be case sensitive. Note that the special characters are codepage sensitive; your terminal may show them differently.

Refer to the reference article for *structure* for the complete syntax of a structure definition.

Activating a Structure Definition

Pass the structure definition to *structure* ADD to define it to *CMS Pipelines*.

Figure 182. Activating a Structure

```
pipe < qacc record | structure add thread
▶Ready;
```

This defines the structure to be available until it is manually deactivated.

Note that the structure definition is the input stream to *structure*; it need not come from a file; it could have been generated by a conversion utility upstream in the pipeline.

You can even annotate the structure definition, as long as you remove your comments before passing the record to *structure*.

You can see the definition of a structure by *structure* LIST:

Figure 183. Listing a Structure

```
pipe structure list qacc | cons
▶:qacc      <length 28>
▶ Mode      1.01
▶ Extn      3.01
▶ Stat      10.01
▶ Files     11.10
▶ Vdev      23.04
▶ Label     Length * <at 29>
▶ Labelc(*) 29.01
▶Ready;
```

You can also obtain a summary of all defined structures:

Figure 184. Listing a Summary of Defined Structures

```

pipe structure listall | cons
▶Thread
▶:qacc      <length 28>
▶Ready;

```

Referencing Fields in a Structure

Figure 185 shows how to select the lines that describe an accessed minidisk, excluding those that reference a directory.

Figure 185. Referencing a Symbolic Field

```

pipe cms query accessed | drop 1 | ...
... pick member qacc.Vdev /!= /DIR / | cons
▶A      R/W      156 191  SRV191
▶R      R/O      893 592  TCM592
▶S      R/O      701 190  MNT190
▶X      R/O      893 120  TCM592
▶Y/S    R/O      1121 19E  MNT19E
▶Ready;

```

We referenced the field “fully qualified” in this example. Were you to reference several fields, you can specify the structure name with the keyword `QUALIFY`, as shown in the slightly contrived example in Figure 186.

Figure 186. Using a Qualifier

```

pipe cms query accessed | drop 1 | ...
... pick qualify qacc m Extn /!= / / & m Stat == /0/ | cons
▶Y/S    R/O      1121 19E  MNT19E
▶Ready;

```

Both `QUALIFY` and `MEMBER` may be abbreviated down to one letter.

You may also specify the option `QUALIFY` to define a default qualifier for all stages of the pipeline specification, but the option must be spelt in full; no abbreviation is available.

Figure 187. Using the `QUALIFY` Option

```

pipe (qualify qacc) cms query accessed | drop 1 | ...
... pick m Extn /!= / /|pick m Stat == /0/ | cons
▶Y/S    R/O      1121 19E  MNT19E
▶Ready;

```

Using Typed Data

The definition of a member may associate a type with it. The type is a single letter in upper case, except for `L`. The type is ignored in general, but *pick* and *spec* support these types:

Structured Data

:	C	A character string.
:	D	Binary integer (big endian) in two's complement notation. The input field may have any length.
:	F	System/360 hexadecimal floating point. The input field may have any length, but only the first sixteen bytes are used (corresponding to extended precision). If it is present, the eighth byte is ignored; it is the characteristic of the lower half.
:	P	System/360 packed decimal integer. A scale may optionally be associated with the member by specifying a signed number in parentheses. A positive scale specifies the number of decimal places; a negative one specifies the number of integer digits to drop on the right.
:	R	Byte-reversed (little endian) binary integer in two's complement notation. The input field may have any length.
:	U	Unsigned binary integer. The input field may have any length.
:		A blank type, which means no type defined, as well as other letters are treated the same as character strings.

Using Arrays

A member of a structure can be defined as an array of fixed or variable dimensions. Such a member is referenced using a subscript in parentheses after the member name. The subscript must be a positive number, except that *spec* is able to use computed subscripts in some contexts.

Our example structure has a slightly contrived array on top of the directory name. To select the second member of this array:

Figure 188. Accessing a Member of an Array

```
pipe cms query accessed | substr member qacc.Labelc(2) | take 3 | ...  
... console  
▶a  
▶R  
▶F  
▶Ready;
```

The entire array is selected when you specify a member that is an array without providing a subscript.

Deactivating a Structure Definition

Once you are done with the structure, you should remove it from *CMS Pipelines* by passing the name to *structure DELETE*, as shown in Figure 189 on page 95.

Figure 189. Deactivating a Structure Definition

```

pipe literal qacc | structure delete thread
▶Ready;

pipe cms query accessed | pick qualify qacc member Vdev == /DIR / | cons
▶Structure not defined: qacc
▶... Issued from stage 2 of pipeline 1
▶... Running "pick qualify qacc member Vdev == /DIR /"
▶Ready(01392);

```

The example also shows that the structure is no longer known to *CMS Pipelines*.

Structure Scopes

You can define structures in caller, set, or thread scope. A number of structures are predefined in *CMS Pipelines*; they are in the built-in scope. Structure names are resolved in the order caller, set, thread, built-in.

Use caller or set scope for production strength applications, unless they are run by an EXEC that contains multiple PIPE commands (you may consider changing that to a single PIPE that runs a REXX program to issue the multiple pipelines with CALLPIPE). Thread scope is appropriate for interactive use. Refer to the usage notes for *structure* for further details.

Caller Scope

Structures defined in caller scope must be defined by a CALLPIPE specification, which logically makes the structures local to the stage issuing this subroutine pipeline and all its descendants. The scope is dismantled when the stage that issued the CALLPIPE terminates. Structures defined in caller scope obscure, for its callees only, all structures of the same name in all other scopes.

There can be any number of caller scopes within a pipeline set. In general, the caller scopes form a forest that has the pipeline set at its root.

Set Scope

Set scope is the default. Structures defined in set scope last until the end of the current PIPE command or until the current record being processed by *runpipe* is consumed. At that point the pipeline set is dismantled and all its contents are discarded, including structure definitions in set scope.

Structures being defined in set scope can embed any already defined structure, except for structures defined in a caller scope in the innermost pipeline set.

A new set scope is established on a recursion into *CMS Pipelines* and by *runpipe*.

A structure defined in set scope temporarily obscures a structure by the same name in all nesting pipelines, in thread scope, and built in.

Structured Data

: Thread Scope

: Thread scope is effectively permanent. Structures defined in thread scope remain defined
: until the end of the CMS process or the z/OS task.

: Structures in thread scope are removed only by explicit *structure* DELETE.

: Structures in thread scope can embed only structures in thread scope and built-in scope.

: Built-in Scope

: Built-in structures are searched last when a structure name is resolved; thus, they may be
: obscured by structures you define, but they cannot obscure a structure defined by you.
: You can reference built-in structures freely in structure definitions; you can list the
: contents of a built-in structure using *structure* LIST; you can list the names of the built-in
: in structures using *structure* LISTALL BUILTIN; but you can neither add nor delete a built-in
: structure.

: ***CMS Pipelines Structures:*** EVENTRECORD FPLASIT FPLSTORBUF

: ***CP Structures:*** VMCMHDR VMCPARM

: ***CMS Structures:*** DIRBUFF

: All built-in structures are caseless.

Chapter 7. Writing a REXX Program to Run in a Pipeline

If you know REXX from writing EXECs or XEDIT macros, you can easily supplement *CMS Pipelines* filters with programs of your own written in REXX. We call such a program a REXX filter or a REXX stage. On CMS, a REXX filter is usually stored in a disk file that has the file type REXX; on z/OS it is usually a member of the data set allocated to DDNAME FPLREXX. It is run by mentioning the file name and arguments as a stage in a pipeline specification. Commands issued from a REXX filter are processed by *CMS Pipelines* (just as commands issued from an XEDIT macro are handled by XEDIT). Pipeline commands read and write the pipeline and perform other functions.

On CMS, the maximum nesting of 200 CMSCALLS does not apply to REXX filters; you can have as many REXX programs running in a pipeline as you have virtual storage for. On z/OS, REXX filters run in separate reentrant environments. There is a predetermined maximum number of possible concurrent REXX environments in an address space. The installation can set this number.

Each REXX stage is a separate REXX program. It has its own set of variables which are distinct from all other variables in all other invocations of REXX programs; other REXX programs run without disturbing a program's variable pool.

A REXX filter reads its input stream(s) and writes its output stream(s) as and when it chooses; the program decides when its task is complete and when it should exit. The pipeline dispatcher runs the pipeline stages so that data move through the pipeline. When a filter reads a record, the dispatcher often turns around and runs some other stage so that it in turn can produce the record to be read.

See Chapter 25, "Pipeline Commands" on page 750 for a reference of all pipeline commands.

Concentrate on getting a simple program working first. Wait with complex programs until you understand the environment REXX filters run in.

Reading and Writing the Pipeline

Figure 190 shows a sample filter that reads no input and produces a single line of output. The first line of the program is a comment (all REXX programs must begin with a comment). The second line issues a command to write a line to the pipeline.

Figure 190. HELLO REXX, a Simple REXX Filter with Usage

```
/* HELLO REXX: REXX filter */
'output' 'Hello, World!'

  pipe hello | console
  ►Hello, World!
  ►Ready;

  pipe hello | xlate upper | console
  ►HELLO, WORLD!
  ►Ready;
```

Figure 191 on page 98 shows the basic copy filter. Add instructions to it to build a filter processing a data stream.

REXX Filters

Figure 191. COPY REXX Copies Input to Output

```
/* COPY REXX -- Copy unchanged */
signal on error
Do forever
  'readto record'
  'output' record
End
error: exit RC*(RC<>12)
```

The pipeline command READTO reads from the pipeline. The argument (record) is the name of the variable that receives the contents of the next record. The assignment is a side effect of issuing the pipeline command, as is setting the variable RC to the return code. This is why the name is a literal inside the quotes. A record is discarded if READTO is issued without an argument.

OUTPUT writes the argument string as a record to the pipeline. An expression is evaluated by REXX before the pipeline command is processed by *CMS Pipelines*. Note the difference between READTO and OUTPUT: the latter has the record to write as its argument string; the former has the name of a variable as its argument.

! Return code 12 on a READTO or OUTPUT pipeline command means end-of-file. When
! READTO receives end-of-file it indicates that no more records will be coming. The
! end-of-file on OUTPUT means that the remainder of the pipeline does not want to receive
! any more records from this stage. In both situations there is no need for the filter to
! continue, so it can simply terminate.

! Because of the signal on error, a pipeline command that sets a nonzero return code
! causes REXX to transfer control to the error: label where the exit statement terminates
! the program.

! The motivation for using signal on error is to avoid an explicit test on return code after
! each pipeline command (and not risk forgetting it).

! For many filters it is not an error when the pipeline decides to stop passing more records.
! It simply indicates that this filter is done with its task and can stop processing records as
! well. Terminating with a return code of 0 indicates that this filter ended without an error.

! The exit statement like in Figure 191 with the computed return code is often used in
! REXX filters as a compact notation for the following statement.

```
! if RC = 12 then exit 0  
! else exit RC
```

! A REXX filter using only the READTO and the OUTPUT pipeline commands is suitable for a
! pipeline specification that contains just one pipeline, but it has the potential to “delay the
! record” (see “Keep the Order of Records” on page 89). Such a delay can lead to unex-
! pected results in a multistream pipeline network; thus we recommend that you learn to
! write robust REXX filters from the beginning. Figure 192 on page 99 shows a copy
! program that does not delay the record.

Figure 192. COPY Program that Does not Delay the Record

```

/* COPYND REXX -- Copy without potential to delay          */
Signal on novalue

'eofreport all'          /* Propagate EOF backwards too */
signal on error

do forever
  'peekto line'          /* Look for next input line    */
  /* Process line here */
  'output' line          /* Pass it to the output      */
  'readto'              /* Consume the record         */
end

error: exit RC*(wordpos(RC, '8 12')=0)

pipe literal a line | copynd | console
►a line
►Ready;

```

The PEEKTO pipeline command sneaks a peek at the next input record without consuming it. When control returns after the PEEKTO pipeline command and the return code is zero, the stage that produced the record is now waiting in an OUTPUT pipeline command. You can peek as often as you like; the same record will be shown until you issue a READTO pipeline command to consume the record. The producer can then resume after its OUTPUT pipeline command.

The expression on the exit statement deserves some explanation. It ensures that the program to terminate with return code 0 when the pipeline command set return code of 8 or 12. It is a more compact way to code the following.

```

if (RC=8) | (RC=12) then exit 0
else exit rc

```

Because EOFREPORT ALL is specified in the filter, the PEEKTO sets return code 8 when the output stream of the filter has become unconnected (which means that the remainder of the pipeline has terminated and will not consume any more records from this stage). Since the filter has been notified that no more records are needed, there is no value in processing the next input record or even waiting for one. This not only avoids wasting resources processing records when there is no need for them, it also helps terminate a complicated pipeline topology in an orderly way. The built-in programs follow this same style where applicable.

Figure 193 on page 100 shows a simple variation of the copy filter is a program to prefix its argument string to each record being copied; a task that could also be done using *insert*.

REXX Filters

Figure 193. PFX REXX Prefixes a String to Records

```
/* Copy input to output */
signal on error
Do forever
  'peekto line'
  'output' arg(1) line
  'readto'
End
error: exit RC*(RC<>12)

pipe literal def | literal abc | pfx Data are: | console
►Data are: abc
►Data are: def
►Ready;
```

This can be generalised to perform an arbitrary operation on the record (see Figure 194) where the argument string is an expression computing the record to write to the pipeline.

Figure 194. RXP REXX Is a Generalised Filter

```
/* Compute expression on each input line */
signal on error
Do forever
  'peekto in'
  interpret "'output' (" arg(1) ")"
  'readto'
End
error: exit RC*(RC<>12)
```

Figure 195 shows *rxp* used with REXX built-in functions to manipulate the data stream. Note that the output expression is enclosed in parentheses. This ensures correct operation even when the expression contains a relational or Boolean operator, which has lower precedence than the blank operator used to concatenate the command (OUTPUT) to the string to be written.

Figure 195. Using RXP REXX

```
pipe literal abc | rxp '***' centre(in, 10) '***' | console
►*** abc ***
►Ready;

pipe literal this is a line | rxp words(in) | console
►4
►Ready;
```

Performance of *rxp* improves if the complete loop is interpreted rather than each OUTPUT pipeline command:

Figure 196. Interpreting a Complete Loop

```

/* RXPI -- Compute expression on each input line */
signal on error
interpret,
"Do forever;",
  "'peekto in';",
  "'output' (" arg(1)");",
  "'readto';",
"End"
error: exit RC*(RC<>12)

  pipe literal abc def  ghi| rxpi space(in, 2, '*') | console
▶abc**def**ghi
▶Ready;

```

But as you can see, this performance improvement comes at the price of making the program much harder to read. We recommend that you keep things simple until you really have a performance problem resulting from interpreter overhead.

See also “Building a REXX Program Dynamically” on page 113.

Using Multiple Streams in REXX Filters

A multistream filter uses more than a single input and a single output stream. For example, a multistream filter could read two input streams and produce records for a single output stream. To write and use a multistream REXX filter, you must know how to invoke it; and you must also know how to access the streams. Chapter 5, “Using Multistream Pipelines” on page 74 describes how to write a pipeline with multiple streams. Figure 197 shows how to pass two files to the REXX filter *scmp*:

Figure 197. Invoking a Multistream REXX Filter

```

/* CMPF EXEC */
address command
'PIPE (end ? name CMPF)',
  '?< first file ',           /* Read first file          */
  '| c: scmp ',              /* Pass on primary input    */
  '| console ',              /* Display result           */
  '? < second file ',        /* Read second file         */
  '| c:'                      /* Pass on secondary input  */

```

CMPF EXEC reads the first file into the primary input stream of *scmp*; it reads the second file into the secondary input stream of *scmp*; and it displays the output on the terminal.

When you write a multistream REXX filter, you use PEEKTO, *etc.*, to perform I/O operations just as you would do with a pipeline specification that contains just one pipeline. Use SELECT to specify which stream is used by subsequent I/O commands. SELECT INPUT 1 switches to the secondary input stream for subsequent PEEKTOS and READTOS. SELECT INPUT 0 switches back to the primary input stream. Output streams are selected similarly; for example, SELECT OUTPUT 1. SELECT ANYINPUT switches to any input stream that has a record available; if there is no record available, it waits for one.

scmp, which is shown in Figure 198 on page 102, discards the first run of identical records from the primary input stream and from the secondary input stream. A single record containing the count of discarded records is written to the primary output stream.

REXX Filters

The return code is 0 when both streams are at end-of-file (and thus their contents are identical); it is 4 when one of the streams is at end-of-file; and it is 8 when neither stream is at end-of-file.

Figure 198. Comparing Streams for Identity

```
/* Compare two streams for being identical */
signal on novalue
do i=0 until data.0~==data.1
  call peek 0 /* Load from stream 0 */
  call peek 1 /* ... and from stream 1 */
  If eof.0 | eof.1
    Then leave /* At least one at eof? */
end
'output' i /* Write result */
exit word('8 4 0', 1+eof.0+eof.1) /* Select return code */

peek:
parse arg which
'select input' which /* Select stream */
If RC ~=0
  Then exit -abs(RC)
If i>0
  Then 'readto' /* Discard previous unless first time */
'peekto data.which' /* Sneak a peek */
If RC<0 | find('0 12', RC)=0
  Then exit RC /* Serious trouble? */
eof.which=(RC=12) /* EOF? */
return
```

The interesting part is in the subroutine `peek`. Its argument is the number of the stream to read. The stream is selected with `SELECT`. The previous record is discarded except for the first time, and the next record is loaded into the data variable with `PEEKTO`, which has a peek at the record without consuming it. The return code sets a variable indicating end-of-file. The data variable `data.which` is dropped at end-of-file, which is why the test for end-of-file is performed inside the loop separate from the test for identity.

Controlling Streams

When a REXX filter is specified with secondary streams (or more), these streams will be available to it. The program can read from them and write to them as and when it pleases; it can reference them in subroutine pipelines (to be described later); and it can even throw them away.

In a multistream REXX filter, the `SELECT` pipeline command specifies which stream to read and which one to write. Subsequent `PEEKTO`, `READTO`, and `OUTPUT` pipeline commands will refer to the stream you selected until you select another one. You can select the input stream independently of the output stream; or you can select both input and output with one command. The primary input stream and the primary output stream are selected when the REXX filter starts.

There are two ways to reference a stream:

- By number. The primary stream has number 0 (zero); the secondary stream has number 1 (one); and so on. The numbering is in the order of the occurrences of the label references to the stage in the pipeline specification.

- *By stream identifier.* A stream identifier can be specified with the label definition and with subsequent label references by suffixing a period and the identifier to the label itself (before the colon that ends the label). The stream identifier uniquely identifies a particular stream. It is useful as a symbolic reference. It can be used even for conditional processing, which depends on the presence of a stream identifier. Stream identifiers are optional.

SELECT with a number is slightly faster than using a stream identifier. You can use the STREAMNUM pipeline command to discover which stream number (if any) is associated with a particular stream identifier and then use the stream number from there on.

Use SELECT ANYINPUT to select whichever stream has a record available. If more than one stream has a record available when you issue the SELECT ANYINPUT pipeline command, it is unspecified which input will be selected.

Use SELECT OUTPUT to select the stream to which OUTPUT writes its argument string. SELECT BOTH selects a stream at both the input and the output side. SELECT ANYINPUT selects whatever input stream has a record available; STREAMNUM INPUT sets the return code to the number of the stream that SELECT ANYINPUT has selected.

Use the MAXSTREAM pipeline command to determine the highest stream number available. Thus, MAXSTREAM returns 1 when you have secondary streams, but not tertiary streams.

Using CALLPIPE to Run a Subroutine Pipeline

CALLPIPE calls a *subroutine pipeline*. This is a handy way to create a synonym for a cascade of filters.

Write a pipeline specification after the command verb (see Figure 199). This particular subroutine pipeline issues CP commands and translates the response to lower case. Use *cplower* whenever you wish a cascade of *cp* and *xlate*.

Figure 199. Subroutine Pipeline		
/* CPLOWER REXX, a Sample Subroutine Pipeline		*/
'callpipe',	/* Command verb	*/
:',	/ Read input stream	*/
cp' arg(1),	/* Issue CP commands	*/
xlate lower',	/* Make lowercase	*/
:',	/ Write output stream	*/
exit RC		

The REXX program waits for all stages of the new pipeline to complete before it continues. The variable RC is set to the “worst” return code from any of the stages.

Specify where to connect the input and output streams of the running stage to the new pipeline with connectors of the form “*:". For simple subroutines, put one of these at each end of the pipeline specification to indicate that the new pipeline should be connected to the currently selected streams.

Using PEEKTO in Figure 198 on page 102 to see the record without consuming it means that the program can be called as a simple front end to a more sophisticated compare program. This is because the records that do not match stay in the producer’s output stream and can be read again, for instance by a control stage.

Figure 150 on page 71 shows another subroutine pipeline.

Sipping at Data—Processing the Input File Piecemeal

Often the input file consists of several subsections that are separated by some particular record. A subroutine pipeline can quickly discard or copy data up to the first or next occurrence of such a delimiter record. To pass the part of a Script file up to the body of the document:

Figure 200. *Sipping at Data*

```
/* FIXSCR REXX -- Process a Script file */
'callpipe *: | tolabel :body.| *:'
```

This subroutine pipeline copies records until *tolabel* reads a record that contains *:body.* in the first six columns. *tolabel* then terminates without consuming the record. This causes end-of-file to propagate from within the subroutine pipeline towards the outside. The input and output streams are reconnected to the REXX program at this point, and the program can now process the body of the file.

Short Circuits

A subroutine pipeline with two connectors and no stages short circuits the streams; that is, it connects the two neighbour stages as if the stage that issued the CALLPIPE pipeline command were not there. That is, records are passed from the neighbour to the left directly to the one to the right. The calling stage waits while records fly overhead and resumes when end-of-file is reflected, at which time the output stream is connected back to the neighbour on the right.

One use of this is to write a variation of *literal* where the literal record is written after the input stream is copied to the output. You could have written a loop to copy the stream but the short circuit is simpler and faster. Figure 201 shows LITAFTER REXX, the REXX formulation of *append literal*.

Figure 201. *LITAFTER REXX Writes Literal after File Is Copied*

```
/* Write literal after the input is copied to the output */
signal on novalue
'callpipe (name litafter) *:|*:' /* Copy the file. */
if RC=0
  then 'output' arg(1) /* Write literal text */
exit RC
```

Use the pipeline command SHORT, rather than a short circuit pipeline, when you are not going to write to the output stream after the input stream has been copied to it. (SHORT is more efficient than the short circuit subroutine.) *literal* can be formulated in REXX using SHORT (see Figure 202 on page 105).

Figure 202. LITERAL Written in REXX

```

/* LITERAL in REXX */
'output' arg(1)
If RC=0
  Then 'short'
exit RC

```

Accessing REXX Variables

The built-in programs *rexxvars*, *stem*, *var*, *vardrop*, *varfetch*, *varset*, and *varload* can be used in a subroutine pipeline (a pipeline specification that is issued with the CALLPIPE pipeline command) to access variables in the REXX program's variable pool. See "Accessing Variables" on page 36.

Obtaining the Source String

To display the current REXX environment (assuming the command is issued to XEDIT and that the file PIPE XEDIT exists and issues the pipeline specification to CMS):

Figure 203.

```

pipe rexxvars | take 1 | console
►s CMS COMMAND PIPE XEDIT * PIPE XEDIT

```

rexxvars writes the source string in its first output record. It then writes two records for each variable in the environment. These lines are discarded in the example above, because *take* only copies the first record. (*rexxvars* is smart enough to detect that its output is being discarded and terminates quickly.)

Scanning the Argument String

While the REXX language features the powerful Parse instruction, which can be used to parse most *CMS Pipelines* syntax variables directly, it does not parse the way *CMS Pipelines* built-in programs scan for an *inputRange* or a *delimitedString*.

As an example, consider the syntax variable *delimitedString*. When a built-in program scans an argument string for such syntax variable, it calls a subroutine, to which it supplies a pointer and a count. The scanning routine then returns a pointer and a count, which represent the delimited string. It also updates its input parameters to reflect what remains to be scanned in the argument string.

Making such a scanning routine available to the REXX filter programmer ensures consistency between built-in programs and REXX filters.

The scanning routines are called through pipeline commands from a REXX filter; all parameters are specified in the command string and all results are fed back through variables, which are set as a side effect.

Thus, the argument string on the scanning pipeline commands consists of three parts, each of which are separated by a single blank.

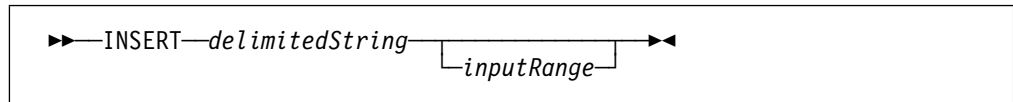
1. Keywords, where required.
2. Literal variable names for the result and the residual string. *CMS Pipelines* sets the variables as a side effect of the pipeline command.

REXX Filters

3. The input string or record to be scanned or processed.

Using this model, *CMS Pipelines* supports pipeline commands to scan the argument string and to get the contents of an input range in a record.

As an example, consider a simplified version of the *insert* built-in program with this syntax:



That is, *insert* requires a *delimitedString* (such as `/abc/` or `x010203`); an *inputRange* (such as `word 3`) is optional after the string.

Figure 204 shows the beginning of a REXX filter that scans its argument string according to the syntax diagram above:

<pre>parse arg args 'scanstring string rest' args /* Get the string */ If RC≠0 /* Bad string? */ Then exit RC 'scanrange optional range rest' rest /* Get the range */</pre>

The first command scans the beginning of the argument string for a delimited string. The result is two strings, the one scanned and the remaining argument string after the delimited string. These are stored into the variables `string` and `rest`, respectively.

The second command scans what remains after the delimited string for an *inputRange*. That is, it determines the position in the input record where the string should be inserted. The keyword `OPTIONAL` specifies that an omitted range should be treated as the complete record. Except for the keyword, `SCANRANGE` is similar to `SCANSTRING`. The range is stored into the variable `range` and the remaining string is stored into `rest`. (Note that REXX referred to the value of the variable when it built the command string; you can reassign its value, just as you can in the REXX Parse instruction.)

Both the `string` and the `range` variables contain the result of the scanning routine, but whereas you can use the `string` directly in REXX (and no doubt you will), the `range` is a “token” in the original English meaning of the word. It is something that *CMS Pipelines* has given you to hold for a while; when you wish to refer to the part of an input record that is defined by this particular input range, you hand the token back to *CMS Pipelines*.

Getting a Range from an Input Record

If you have used the `SCANRANGE` pipeline command to obtain a token representing the specification of an input range, you can obtain the data in the range with the `GETRANGE` pipeline command, as shown in Figure 205 on page 107:

Figure 205. Using GETRANGE

```

signal on error
do forever
  'peekto line' /* Get some input */
  'getrange range stem parts.' line /* Split it up */
  If parts.0=1 /* Was range in record at all? */
    Then 'output' string || line /* No, just insert first */
    Else 'output' parts.1 || string || parts.2 || parts.3
  'readto' /* Consume record */
end
error: exit RC*(wordpos(RC, '8 12')=0)

```

The arguments to the GETRANGE pipeline command consist of three words and a string. The first word specifies the name of the variable that contains the token that represents the input range; this variable was set by SCANRANGE. The second word is a keyword; it specifies that you wish the result in a stemmed array. The third word specifies the stem of the compound variables into which the result is to be stored. The remainder of the command (after exactly one blank) is the input line from which the input range is to be extracted.

The variable stem.0 is set to one or three. It is set to one when the input range is not present in the record; stem.1 is then set to the entire record. When stem.0 is set to three, the part of the input record up to the beginning of the range is stored into stem.1; the contents of the range are stored into stem.2; and the part of the input record after the range is stored into stem.3.

Building Production Strength REXX Filters

There are many considerations for writing a production strength filter. You should ensure that:

- It scans its arguments in the *CMS Pipelines* tradition.
- It issues a message when it discovers an error.
- It uses the COMMIT pipeline command after it has scanned its arguments, before it begins to process data.
- It does not delay the record unnecessarily.
- It propagates end-of-file in both directions.
- It enforces particular requirements, such as having only primary streams, always having secondary streams.

Scanning Arguments

There is one more twist to scanning arguments. Suppose you want to pass part of the argument string to a built-in program, for example, an *inputRange*, you should use the scanning commands already described in “Scanning the Argument String” on page 105, but the resulting token is not of much use.

Instead, you can infer the input string that was scanned by the length of the residual string:

REXX Filters

Figure 206. Finding String to Pass to a Built-in Program

```
parse arg argstring
'scanrange required token rest' argstring
range=delstr(argstring, length(argstring)-length(rest)+1)
'callpipe ... |spec' range ' 1 ...'
```

Be careful when you use part of the argument string in a pipeline specification; the string may contain a stage separator or an end character, which will need to be doubled up to be escaped. The probability of this happening in an *inputRange* is low, but not zero; still, you may wish to accept this restriction. (For example, the word separator could be specified as a vertical bar.)

When scanning for a *delimitedString*, however, the exposure is real, but the cure is different. The idiomatic way to pass a delimited string to a stage is shown in Figure 207. The trick is to convert the string to hexadecimal notation, which means that the string in the pipeline specification will not contain any special character at all.

Figure 207. Passing a delimitedString to a Stage

```
parse arg argstring
'scanstring string rest' argstring
'callpipe ... | insert x'c2x(string) '|...'
```

Issuing Error Messages

To issue an error message, your immediate reaction might be to use the Say instruction, but that has none of the advantages of the methods to be described; and it has all of their disadvantages.

CMS Pipelines provides two pipeline commands to issue messages: MESSAGE and ISSUEMSG; and CMS has the XMITMSG command.

The MESSAGE pipeline command simply writes its argument string to the current message disposition. This is usually your terminal, but see also *runpipe*. You must supply the entire message, including the message prefix, for example:

Figure 208. Using the MESSAGE Pipeline Command

```
'maxstream input' /* How many streams have we? */
if RC>1
  then 'message INSERT264E Too many streams.'
```

The advantage of the MESSAGE pipeline command is its simplicity. The disadvantage is that you have the message text deep in your code, which makes it difficult to change and almost impossible to support translation to national languages (NLS) and multiple message repositories.

The ISSUEMSG pipeline command is an interface to the message infrastructure in *CMS Pipelines*. The argument string contains the message number, the module identifier, and as many delimited strings as there are substitutions in the message:

Figure 209. Verifying Multistream Configuration

```

'maxstream input'          /* How many streams have we? */
If RC>1                    /* More than two? */
  Then call err 264        /* Say so */
If RC=0                    /* Or too few? */
  Then call err 222        /* Say so too. */
'streamstate output 1'    /* Is secondary output connected? */
If RC≠12
  Then call err 1197

```

Setting things up for ISSUMSG is best done in a subroutine, as shown in Figure 210. The first argument string contains the message number; subsequent argument strings contain strings to be substituted:

Figure 210. Subroutine to Issue Messages

```

err:
parse arg msgno .
sub=''
Do i=2 to arg()
  sub=sub '00'x || translate(arg(i),, '00'x) || '00'x
End
parse source . . myfn .
trace off                      /* Be quiet on MVS */
'issuems' msgno myfn sub
exit RC

```

The advantage of ISSUMSG is that it uses the *CMS Pipelines* infrastructure and thus allows for NLS to the extent that *CMS Pipelines* does. The disadvantage is that it may be cumbersome to add your own message to *CMS Pipelines*'s repository. Prior to level 1.1.10/0015 it would entail making a filter package to contain the repository. From 1.1.10/0015, you can add your messages to the FPLUME REPOS repository and install this user repository. (See Chapter 28, "Configuring *CMS Pipelines*" on page 867.)

Use the CMS command XMITMSG to issue a message using a CMS message repository. This has the advantages of using standard message repositories, but the disadvantage that the message will be written to the terminal irrespective of the *CMS Pipelines* message disposition.

Using the COMMIT Pipeline Command to Ensure other Stages Are Committed to Process Data

Use the COMMIT 0 pipeline command when the arguments have been processed without errors and the program is about to start processing data. If the return code on COMMIT is nonzero, it means that some other stage has found an error and has terminated with a nonzero return code. Thus, the pipeline will be abandoned; and the program might as well terminate at this point.

REXX Filters

Figure 211. Using the COMMIT Pipeline Command

```
'commit 0'                /* We're ready to process data */
If RC~=0                  /* But someone else isn't!    */
    Then exit 0
```

If your program has allocated resources before committing, it must deallocate those resources when it discovers that the pipeline is being abandoned, just as it must when it terminates normally.

Propagating End-of-file

You should make an effort to avoid unnecessary processing. For example, if an output stream has been severed by its consumer, there is no point in producing output on it once you know that it has been severed. And when you realise that all output streams are gone, you should terminate unless you can do useful work (which would imply that your REXX filter acts as a device driver rather than as a true filter).

So how do you realise that it is time to call it quits? You may wish to check the status of your streams from time to time. You can do this in several ways:

- Issue the STREAMSTATE pipeline command to determine the state of a particular stream. You can loop over all defined streams to get the whole picture. This is a bit cumbersome.
- Issue the STREAMSTATE ALL command with the name of a variable to be set. *CMS Pipelines* then stores the status of all defined streams into this variable. You still need to write a loop to process the status of the individual streams.
- Issue the STREAMSTATE SUMMARY pipeline command to get a return code that can be used directly for your decision. If the return code is 8, either all inputs are gone or all outputs are gone.

But even with these very sharp tools, you are still not able to emulate what the built-in programs can achieve. The remaining problem is to discover that an output stream has been severed *while* the REXX program is waiting for an input record. Once you have issued the SELECT ANYINPUT or the PEEKTO pipeline command, you will not get control until a record is available (or there is end-of-file on the input).

The solution is the EOFREPORT pipeline command, which modifies the semantics of the commands to read and write. Issue the EOFREPORT ALL to be alerted when all output streams have been severed while you are waiting for an input record. The return code will be 8 when all outputs are gone. You should issue this command in all your production strength REXX filters.

Figure 212. Using EOFREPORT

```
trace off                /* Don't show errors          */
'eofreport all'          /* Propagate EOF              */
```

If the EOFREPORT pipeline command is issued to a version of *CMS Pipelines* that does not support the command, the return code will be -7. You can ignore this error. Since REXX traces negative return codes by default, you should turn trace off to avoid a nuisance message in this case.

But even this is not always enough. Sometimes you may wish to propagate end-of-file on individual streams, as is done, for example, by *gate*. Issue EOFREPORT ANY to be alerted to any change of pipeline connections while you are waiting for a record. Return code 4 means that some as yet unknown stream has been severed; you must issue STREAMSTATE ALL and parse the variable it sets to discover which stream(s) need severing.

With EOFREPORT ANY we are speaking fine detail. Even an OUTPUT pipeline command will terminate with return code 4 when a stream has been severed before the consuming stage has seen the record. (That is, you can in some circumstances produce an output record and then later retract it!) If the consuming stage has seen the output record, it is too late. The producing stage must remain blocked (it cannot be resumed) until the record is consumed or the consumer severs the stream.

Note that return codes 4 and 8 are set only when the stage is blocked at the time the stream is severed. If a stage is ready to run, but not dispatched, there will be no indication that a stream has been severed, because the pipeline dispatcher can reflect only one return code at a time.

A Complete Robust REXX Filter

Figure 213 on page 112 shows the complete REXX filter from which the examples were taken in the previous sections:

REXX Filters

Figure 213. Robust REXX Filter

```

/* Insert a string in a record. */

'maxstream input' /* How many streams have we? */
If RC>1 /* More than two? */
    Then call err 264 /* Say so */
If RC=0 /* Or too few? */
    Then call err 222 /* Say so too. */
'streamstate output 1' /* Is secondary output connected? */
If RC~=12
    Then call err 1197

parse arg args

'scanstring string rest' args /* Get the string */
If RC~=0 /* Bad string? */
    Then exit RC
'scanrange optional range rest' rest /* Get the range */

If rest='' /* Too much? */
    Then call err 112, rest /* Go complain */

'commit 0' /* We're ready to process data */
If RC~=0 /* But someone else isn't! */
    Then exit 0

trace off /* Don't show errors */
'eofreport all' /* Propagate EOF */

signal on error
do forever
    'peekto line' /* Get some input */
    'getrange range stem parts.' line /* Split it up */
    If parts.0=1 /* Was range in record at all? */
        Then 'output' string || line /* No, just insert first */
        Else 'output' parts.1 || string || parts.2 || parts.3
    'readto' /* Consume record */
end
error: exit RC*(wordpos(RC, '8 12')=0)

err:
parse arg msgno .
sub=''
Do i=2 to arg()
    sub=sub '00'x || translate(arg(i),, '00'x) || '00'x
End
parse source . . myfn .
trace off /* Be quiet on MVS */
'issuemsg' msgno myfn sub
exit RC

```

Building a REXX Program Dynamically

rexx can read the program to run from an input stream and then provide this program to the interpreter rather than let the interpreter find the program to run. The program could be compiled, but that does not seem to be of practical application; this section focuses on interpreted programs.

The program can be read from any input stream; this stream will be at end-of-file when the program starts. Instead of using the Interpret instruction, the program in Figure 194 on page 100 can be generated in a subroutine pipeline like this:

Figure 214. Generating a REXX Filter Dynamically

```

/* RXPD -- RXP dynamically */
pgm.1="/* RXP */ Signal on error"
pgm.2="Do forever; 'peekto in'"
pgm.3="'output' (" arg(1) ")"
pgm.4="'readto'; end"
pgm.5="Error: exit RC*(RC<>12)"
pgm.0=5
'callpipe (end ?)',
  '|*:', /* Read input */
  '|r: rexx (*.1: rxp)', /* Run program */
  '|*:', /* Write output */
  '?stem pgm.', /* Get program */
  '|r:' /* Feed to secondary */
exit RC

pipe literal oscar | rxpd "*"in*" | console
►*oscar *
►Ready;

```

The argument to the *rexx* stage specifies that it should read the program from the secondary input stream (*.1:) and that the file name returned by the Parse Source instruction should be RXP.

Note that each input record becomes a line of the program; semicolons separate REXX instructions on a line.

Implementing a REXX Macro Processor

Think of an XEDIT macro or an ISPF/PDF macro written in REXX. The macro issues commands to the editor which in turn sets the return code and other variables in the macro's variable pool. The macro and the editor really are *coroutines*: each maintains a separate state, and they take turns at performing the task at hand. Macro processors are implemented by command recursion on CMS and TSO because these systems do not support coroutines; a more natural way to implement a macro processor is as a pipeline stage where input records are interpreted as commands. To do this, the macro processor must be able to set variables in the macro's variable pool. These *CMS Pipelines* features support writing macro processors:

- The ability to read a REXX program from a stream means that the macro can be prefixed by REXX statements that set up for specific processing; for instance, a BEGOUTPUT pipeline command can be issued to bring the macro into continuous output

REXX Filters

mode. The macro can also be transformed; for instance, to remove Address instructions.

- The BEGOUTPUT pipeline command, when issued in the macro, has the effect that subsequent commands are written directly to the output stream rather than being processed by *CMS Pipelines* as pipeline commands. Thus, what the user writing the macro thinks of as commands is passed to the following stage as data records.
- The macro processor can use the PRODUCER option on the device drivers for REXX variables (for instance *stem*) to access the REXX variable pool of the macro rather than its own variable pool.
- The SETRC pipeline command allows the macro processor to set the return code in the macro.

It is certainly possible to write an editor that supports XEDIT macros to process data in the pipeline. Such a REXX filter will (if written correctly) be directly transportable between CMS and TSO. It should also be possible to write an XEDIT macro processor to allow XEDIT macros to be used with ISPF/PDF.

Miscellaneous Issues

Issuing Commands from a REXX Filter on CMS

Before explaining how to issue host commands from a REXX filter, a word of warning.

If you issue a command directly from a REXX filter to the host system, *CMS Pipelines* has no way to know that you have given control over to the host system. In many cases, this makes no difference, but there are two pitfalls you should try to avoid:

1. The time spent in the host is charged to the stage by RITA. After all, *CMS Pipelines* does not know that you have given control over to the host, how could it tell Rita?
2. On CMS, the *delay* stage cannot recover from a program that uses the clock comparator.

Instead of addressing commands directly to CMS, use the *command* stage in a subroutine pipeline to issue them when you are not sure whether the command will interfere with *CMS Pipelines* or not. (Or use *subcom* CMS.) This lets *CMS Pipelines* in on what is going on and it may keep you out of trouble.

Note that a REXX filter has no way in general to discover that a pipeline is being timed or that the pipeline contains a *delay* filter. Therefore, if you are writing production strength REXX filters, avoid the use of the address instruction if you can. One severely burnt plumber wrote:

```
Me? I'm just going to NEVER AGAIN use Address COMMAND in a REXX
filter - if it occasionally costs me a few extra microseconds, then
so be it.
```

Now that you have been duly warned, let us discuss how to address commands to particular environments in a REXX filter.

In a REXX filter, you can use the Address instruction to issue a command to other command environments on CMS. As an example, on CMS, this clears the screen before writing a line:

Figure 215. CMS Command from Filter

```

/* TOPLIT REXX */
address command 'VMFCLEAR'          /* Clear screen          */
'output At top of screen?'          */

```

When a subroutine returns, REXX restores the default command environment to the one in effect when the subroutine (or function) was called. Thus, it is safe to change the default command environment in a subroutine that does not issue pipeline commands and does not call a subroutine that issues pipeline commands.

Figure 216. Clearing the Screen

```

/* Clear screen when there are data          */
signal on novalue                          */
'commit 0'                                  /* See if we're running  */
If RC/=0                                    /* not today             */
    Then exit                               /* Get a record          */
'peekto'                                    /* No file?             */
If RC/=0                                    /* clear the screen now  */
    Then exit                               /* Copy data            */
call clear                                  /* COMMAND environment  */
'short'                                     /* Clear screen         */
exit RC                                     /* OK?                  */

clear:
address ''                                  /* COMMAND environment  */
'VMFCLEAR'                                  /* Clear screen         */
If RC=0                                     /* OK?                  */
    Then return
exit RC

```

Do not change the command environment permanently unless you know how to get back. It is safer to issue all commands to other environments with the Address instruction so that you are sure that you retain the pipeline command environment. It is not recommended to use the Address instruction without operands to toggle between command environments.

Issuing Commands from a REXX Filter on TSO

On z/OS, REXX filters run in what is called *reentrant environments* to enable them to terminate in any order. This means that the only command environments available are the pipeline command environment, LINK, z/OS, and similar. In particular, neither TSO nor the ISPF environments are available through the Address instruction. But you can reach them anyway; there are three ways:

- Use *command* to issue a TSO command and have the response displayed directly on the terminal.
- Use *tso* to issue a TSO command and write the response into the pipeline.
- Use *subcom* to reach other command environments (for example ISPEXEC).

The initial command can be specified as the argument; additional commands are read from the input.

Issuing Pipeline Commands from an External Function

In a REXX filter, use the pipeline command `REXX` to call an external REXX procedure that issues pipeline commands. This is equivalent to calling an external function with one argument string. The return code is stored in the variable `RC`, as for all commands. The REXX pipeline command does not support multiple argument strings; a program called this way can return only a number.

Red Neon!

You cannot issue pipeline commands from a REXX program that has been called as an external function or invoked with the `Address` instruction because this implies a `CMSCALL`. Do not pass the result of `Address()` to an external function; results are unpredictable if you issue pipeline commands from REXX programs that are neither invoked as stages nor called by the REXX pipeline command from a stage. You are likely to encounter a disabled CMS wait when the REXX program tries to return.

Return Codes -3 and -7

You receive these return codes when a command is not recognised by the command environment to which it is addressed. This can be because you have addressed the command to the wrong environment or because a continuation character is missing.

CMS and REXX on TSO give return code -3 when they do not recognise the command you have issued to them. This is likely to happen when you issue a pipeline command to the host. Be sure you keep the original command environment intact if you issue the `Address` instruction to select a new environment permanently.

CMS Pipelines gives return code -7 when it does not recognise a pipeline command. The two most likely reasons are:

- A CP or CMS command is addressed to the pipeline command environment. Use the `Address` instruction with `CMS` or `COMMAND` to issue the command to CMS rather than as a pipeline command.
- A continuation comma is missing in a pipeline specification written over several lines.

By default, REXX traces commands that give a negative return code. Since REXX's message is quite explicit, *CMS Pipelines* does not issue further messages in this case. (Thus, if you wish to handle this condition quietly, you can do so.)

On TSO, trace from REXX filters is written to the `DDNAME SYSTSPRT`; be sure to allocate it! Be sure to test the return code or use `signal on error` to trap errors.

Pitfalls

Here are some pointers for when things break inexplicably. Be sure also to read the first section of "Issuing Commands from a REXX Filter on CMS" on page 114.

Calling External Functions from a REXX Filter

REXX supports external functions for REXX filters, just as it does for other kinds of REXX programs, such as XEDIT macros, but be careful not to let REXX's search order trip you over.

When the file type of the REXX program is not EXEC and REXX resolves an external function that is not in a loaded function package, REXX on CMS first searches for a file with a file type like the calling program.

Thus, if you call the external function `myfunction`, REXX looks first for `MYFUNCTI REXX` and then for `MYFUNCTI EXEC`. Thus, if you try to hide an existing EXEC with a REXX filter, it simply will not work; you will get an unending recursion instead. If you are lucky you run out of storage before CMS reaches the limit on nested SVCs and ABENDs you.

On z/OS, REXX searches only the data set from where the calling function was loaded. Thus, you may need to maintain two copies if an external function is called both from a CLIST and from a REXX stage.

The Dangers of Using Implied REXX Filters

The pipeline specification parser looks for a file with file type REXX when it cannot resolve the name of a filter. This allows you to add your own REXX filters seamlessly.

But note that the search for a REXX filter is after the search for a built-in program. If you choose easily understandable names for your REXX filters, it may well happen that a new release of *CMS Pipelines* has a built-in program with the same name as your REXX filter. And then the built-in program “wins”; and your users get frustrated.

The obvious recommendation is to use an explicit *rexx* stage to run the REXX program; this will even save you an infinitesimal amount of CPU time. But if you later decide to incorporate the REXX filter in a filter package, it will not be resolved; the contents of a filter package are effectively built-in programs, even when they are written in REXX.

Thus, for production strength, choose the file names for REXX filters carefully. You might consider prefixing the names with your company's acronym or your own initials; this should reduce the probability of a naming clash.

So what can you do after you have been bitten? Suppose you have a `IF REXX` which begins to fail with the most strange error messages after you install a new level of *CMS Pipelines* (or a new level of CMS). Chances are that *if* is now a built-in program. And you have literally thousands of references to it scattered over hundreds of files. You can give yourself time to think by putting your REXX filter into a filter package that has the magic file name `PIPPTFF`. Filters in this package override the built-in programs.

Still, this may not be a good idea either: Many parts of CMS run pipelines under the covers; if you replace a built-in program with a program of your own by putting it in `PIPPTFF` filter package, this will also affect the pipelines that are written with the real built-in program in mind.

Performance

Remember these points when writing programs for *CMS Pipelines*:

- Make sure the function cannot be performed with a built-in filter or a cascade of such filters. For instance, *spec* should be used instead of the program shown in Figure 193 on page 100.
- Make a program do one thing, and do it well. Decompose a complex task into a suite of simpler generalised programs. Write one small program to perform what is unique to the task.
- Measure, if performance is a concern. Only when convinced of a substantial savings should you contemplate writing a filter to combine function already available in separate programs.

Figure 217. Filter Performance

!	disk dmsgpi maclib count lines cons
!	94814
!	Ok. 11:13:22 02/08/16 0.047 0.056
!	disk dmsgpi maclib pfx prefix count lines cons
!	94814
!	Ok. 11:08:17 02/08/16 2.601 2.618
!	disk dmsgpi maclib spec ,prefix, 1 1-* next count lines cons
!	94814
!	Ok. 11:09:00 02/08/16 0.292 0.301
!	disk dmsgpi maclib change ,,prefix, count lines cons
!	94814
!	Ok. 11:09:24 02/08/16 0.168 0.178
!	disk dmsgpi maclib insert ,prefix, count lines cons
!	94814
!	Ok. 11:09:48 02/08/16 0.136 0.146

Figure 217 shows the difference between a REXX filter and an equivalent built-in program. Words four and five of the ready message show CPU used on a lightly loaded system. The first command shows the overhead of reading the largest file on the system disk and counting the records in it. The second test uses the program shown in Figure 193 on page 100, whereas the next two examples use *spec* and *change* to do the same thing. The final example shows that a special purpose program like *insert* can speed up things even more.

Should You Compile Your REXX Filters?

The performance of filters that do little processing for each record will be dominated by the EXECCOMM processing that moves records into and out of the REXX variable pool. This processing takes roughly the same time for compiled and interpreted REXX; there will be no performance gain from compiling the examples in this chapter. On the other hand, the *cmprrex* utility program showed a marked performance improvement when it was compiled.

Issues other than performance may influence your decision in favour of compiling:

- The compiler may support a higher language level than the interpreter does.
- You may wish to hide the source for a REXX filter so that a user cannot tamper with it.

- The compiler can detect syntax errors in the program that may go unnoticed in testing, because some part of the program is not exercised by the test cases.

We recommend that you compile complex REXX filters even if you decide to use the interpreter when they are run. The compiler finds many errors that would otherwise have gone unnoticed for a while. We have discovered many spelling errors that would not have caused an interpreter diagnostic, by scanning the cross reference listing from the compiler for unreferenced variables and strings that look very much alike (enabled with the XREF option).

MVS Considerations

When *TSO Pipelines* is running a REXX filter, it runs in a reentrant environment. Output to the terminal (for example, from the Say instruction) in such an environment is written to the DDNAME SYSTSPRT rather than directly to the terminal.

Error messages are issued to this data set as well.

Be sure to allocate this data set in your logon procedure or in the Job Control Language for the job step that invokes *TSO Pipelines*. If the DDNAME is not allocated, REXX will issue a message to the programmer to this effect. But you will only see this message in TSO if you have PROFILE WTPMSG.

Thus, if the REXX filter fails and SYSTSPRT is not allocated, you will most likely just see a broken pipe that has no data coming out. This can lead to much head-scratching and finger-pointing.

Be smart: allocate SYSTSPRT!

Chapter 8. Using Pipeline Options

Pipeline options control the pipeline specification parser and the pipeline dispatcher. Options to control the pipeline specification parser are specified in parentheses at the beginning of the pipeline specification. Options to control the pipeline dispatcher apply to all stages when the option is at the beginning of the pipeline specification; options can further be enabled or disabled for a particular stage. You can do this with pipeline options:

- Define characters that have special meaning in a pipeline specification.
- Name the pipeline specification.
- Request additional informational messages.
- Set the message level to enable or disable additional messages that are issued automatically to identify the stage issuing an error message.

Options for the Pipeline Specification Parser

Write pipeline options in parentheses immediately after the command that issues the pipeline specification (before the first stage). A stage separator character is optional after the right parenthesis ending the option list; use one to show without ambiguity that the parentheses contain options that apply to the complete pipeline specification.

Control the Pipeline Specification Parser: The default stage separator is the solid vertical bar that you have by now seen many times. Sometimes you may wish to use the vertical bar as an argument to a filter; for instance to find records starting with a solid vertical bar. There are three ways to do this, shown in Figure 218:

- Use an additional stage separator character for a self-escaping sequence. Two adjacent stage separator characters are treated as a single normal character, which is passed to a program as part of the argument string.
- Define an *escape character* to put in front of solid vertical bars and other characters that are not to be taken as stage separators.
- Use the option SEPARATOR to redefine the stage separator to a different character.

The escape character is defined with the option ESCAPE; when defined, it can be put in front of any character, not just the stage separator. The escape character itself is ignored and any special meaning the following character might have to the pipeline specification parser is suppressed, so the second character becomes just a “normal” one. Use two escape characters to specify a single escape character in an argument string.

Figure 218. Finding Lines Beginning with |

```
/* Self-escaping */
'pipe < some file | find ||| > revised lines a'

/* Define Escape Character */
'pipe (escape ") < some file | find "|| > revised lines a'

/* Change Stage Separator Character */
'pipe (separator ?) < some file ? find |? > revised lines a'
```

All three examples in Figure 218 select records with a solid vertical bar in column 1.

The option ENDCHAR (which is often abbreviated to “end”) is the last of the special characters you can define. It is used to delimit pipelines when using multiple streams. The end character is also self-escaping; use two abutted end characters to provide an end character as the argument to a stage.

The word after the option SEPARATOR, option ENDCHAR, or option ESCAPE is an *xorc*, which is a single character, as we have seen, or a two-character hex value. This is particularly useful in REXX programs where you can use a character that the user cannot type on the terminal. However, remember that it is only the specification of the character in the global option that can use the two-character hex value; it must be a single character in the pipeline specification proper, but with REXX this can be coded as a hex constant. Figure 219 is an example.

Figure 219. Using Two-character Hex Values for the Stage Separator

```
/* Using 01 as the stage separator character */
'pipe (separator 01) < some file' '01'x,
'find |' || '01'x,
'> revised lines a'
```

Parentheses, the asterisk (*), the colon (:), the period (.), and the blank (X'40') have special meaning to the pipeline specification parser; these characters are rejected when used for the scanner characters.

The escape character is not effective when scanning pipeline options; it is not possible to use a right parenthesis for a value of a pipeline option.

Name the Pipeline Specification: The option NAME followed by a blank-delimited word associates a name with a pipeline specification. This name is displayed in messages, but has no other effect. The name need not be unique among the current set of pipelines.

Figure 220. Sample Name

```
pipe unknown | console
▶Entry point UNKNOWN not found
▶... Issued from stage 1 of pipeline 1
▶... Running "unknown"
▶Ready(-0027);

pipe (name test) unknown | console
▶Entry point UNKNOWN not found
▶... Issued from stage 1 of pipeline 1 name "test"
▶... Running "unknown"
▶Ready(-0027);
```

Though of limited use when the pipeline specification is typed at the terminal, the name option is useful in nested subroutine pipelines. If you write the name of the EXEC in the option NAME, *CMS Pipelines* can tell you where there is trouble. FMTP XEDIT automatically inserts the file name and line number as the pipeline name when it converts a pipeline from landscape to portrait format.

Options for the Pipeline Dispatcher

You can specify the options to be described next both in front of the entire pipeline specification (these are *global* options), and in front of individual stages (*local options*). When writing local options, prefix NO to an option to negate the effect of an option that applies to the entire pipeline specification.

Get More Informational Messages: You may wish further information when the PIPE return code is not zero and you get no error message from any stage. Three levels of additional information messages can be requested from the pipeline dispatcher:

- LISTERR Issue a message when a stage returns with a nonzero return code. Use this option to see which stages return “quietly” with a nonzero return code.
- LISTRC A message is issued when a stage is started and when it returns, whether the return code is zero or not.
- TRACE This option causes a large amount of trace data to be written. All calls to the pipeline dispatcher are traced as are its actions. Using this trace you might be able to relate messages issued by other commands or REXX Say instructions to a specific stage.

You can also reduce the number of messages issued:

- MSGLEVEL Enables or disables additional messages that are issued to pinpoint the stage or command that issued a message.

Though it is not exactly providing an informational message, the following option may be useful when you are debugging an Assembler program running in *CMS Pipelines*.

- STOP A message is issued when each stage is started and an address stop (or similar trace) is activated to stop in CP console function mode as soon as the first instruction in the stage is issued. You can set up traces within the program under test. Be sure to have SET RUN OFF when using this option; the stage runs away from you if RUN is ON!

Use *runpipe* as shown in Figure 221 to redirect *CMS Pipelines* messages to a file.

Figure 221. Using *runpipe* to Capture Pipeline Trace

```
/* Trace a pipeline */
pipe='(trace) literal hello|console'
address command
'PIPE',
  ' var pipe',
  '|runpipe',
  '|> pipeline trace a'
```

Specify a local option to activate trace for a single stage of a pipeline, or selected stages. Turning off the rightmost three bits in the message level suppresses the normal messages to identify the stage issuing the message.

Figure 222. Disabling Additional Messages

```
pipe literal abc | (trace nomsglevel 7) literal def | hole
▶FPLDSQ028I Starting stage with save area at X'03B58470 03F290A8 00000000>
▶FPLDSQ001I ... Running "literal def"
▶FPLDSP035I Output 4 bytes
▶FPLDSP039I ... Data: "def"
▶FPLDSQ031I Resuming stage; return code is 0
▶FPLDSP034I "SHORT" called
▶FPLDSQ031I Resuming stage; return code is 0
▶FPLDSP020I Stage returned with return code 0
▶Ready;
```

Chapter 9. Debugging

CMS Pipelines issues a message when it detects an error. But sometimes the mistake is not a syntactical one. A few hints are given below about the things you can do to find out what went wrong when you get no output or get an unexpected return code.

Error Messages

A filter issues an error message when the parameter list is in error or when an error occurs during processing. Figure 223 is a sample run.

Figure 223. Sample Session with Errors

```
pipe console|block 1 vbs|> vb file a
▶Block size too small; 9 is minimum for this type
▶... Issued from stage 2 of pipeline 1
▶... Running "block 1 vbs"
▶Ready(00115);

set emsg on
▶Ready;

pipe console|block 1 vbs|> vb file a
▶FPLBLK115E Block size too small; 9 is minimum for this type
▶FPLSCA003I ... Issued from stage 2 of pipeline 1
▶FPLSCA001I ... Running "block 1 vbs"
▶Ready(00115);
```

Lines are read from the console and blocked with OS record descriptor words (or at least that was the intent). *block* decides it cannot do what is asked and issues error message 115. *CMS Pipelines* adds two messages to help you find the error.

The error in *block* is discovered before any of the stages begin running; no stage is started. After the error in this example, issue “pipe help” to display more information about message 115.

Use *runpipe* to issue a pipeline specification and capture all messages issued from it. This may be a more convenient way to document a problem than console SPOOL. Be sure to use *diskslow* if the problem causes an ABEND. This ensures that the output file will be readable and will contain all records.

Other Hints

Use the option LISTERR to list stages giving a nonzero return code. This lets you find stages that do so “quietly” without issuing an error message.

Figure 224. Effect of LISTERR and NAME Global Option

```

pipe hole|maclib
▶Ready(00012);

pipe (listerr) hole|maclib
▶FPLDSP020I Stage returned with return code 12
▶FPLMSG003I ... Issued from stage 2 of pipeline 1
▶FPLMSG001I ... Running "maclib"
▶Ready(00012);

pipe (listerr name maclib_test) hole|maclib
▶FPLDSP020I Stage returned with return code 12
▶FPLMSG004I ... Issued from stage 2 of pipeline 1 name "maclib_test"
▶FPLMSG001I ... Running "maclib"
▶Ready(00012);

```

Here you see the effect of the option LISTERR. *maclib* gives return code 12 without issuing a message when there is no stage to read its output.

The last sample in Figure 224 shows how to name a pipeline. This is particularly useful when running a pipeline from an EXEC to indicate which EXEC invoked *CMS Pipelines*; knowing which stage issues a message may not be too helpful if one does not know which EXEC contains the pipeline specification being run.

Who Did That?

Use the option TRACE to follow the pipeline dispatcher as it switches control between stages. If you are desperate you may consider running the pipeline through *runpipe TRACE*. This forces trace of *everything*: all subroutines and all pipeline specifications added with ADDPIPE. (Performance is likely to deteriorate.)

Figure 225. RUNPIPE TRACE

```

pipe literal hole | runpipe trace | spill 80 offset 10 | console
▶FPLDSQ1381I Pipeline committed to 0 worst return code 0
▶FPLDSQ028I Starting stage with save area at X'03B586A8 03F2A020 00000000>
▶
▶   commit level 0
▶FPLMSG003I ... Issued from stage 1 of pipeline 1
▶FPLMSG001I ... Running "hole"
▶FPLDSP033I Input requested for 0 bytes
▶FPLMSG003I ... Issued from stage 1 of pipeline 1
▶FPLMSG001I ... Running "hole"
▶FPLDSQ031I Resuming stage; return code is 12
▶FPLMSG003I ... Issued from stage 1 of pipeline 1
▶FPLMSG001I ... Running "hole"
▶FPLDSP020I Stage returned with return code 0
▶FPLMSG003I ... Issued from stage 1 of pipeline 1
▶FPLMSG001I ... Running "hole"
▶Ready;

```

The example in Figure 225 shows how to issue a trivial pipeline (it has only the stage, *hole*) through *runpipe*. Normally, you would build the pipeline specification in a REXX variable and then insert it into the pipeline with a *var* stage.

Debugging

If this still leaves you without a clue, try *runpipe* EVENTS. Refer to Appendix G, “Format of Output Records from *runpipe* EVENTS” on page 939 for the format of the output records. Be prepared to sift through large amounts of data.

No Output

Sooner or later, you run a pipeline and get no output. You can run the pipeline one stage at a time using utility files to notice where it disappears. This may be a fine approach when developing programs for a pipeline, because you can test each stage individually and as cheaply as possible (and to completion) when the input to the stage under test is simply the file generated when the previous stage tested OK. However, to see what went wrong in an existing pipeline, add > stages around selected stages to take a snapshot of the data flying by. When capturing a copy of the data flowing in the pipeline it may be important that end-of-file is propagated backwards through this device driver stage. Use *eofback* to run the device driver, for example:

```
!
!
!
!
... | eofback > trace file1 a | ...
```

Pipeline Stall

A multistream pipeline stalls when, for instance, a stage refuses to produce output.

What happens next depends on the setting of the two configuration variables *STALLACTION* and *STALLFILETYPE*. Refer to Chapter 28, “Configuring *CMS Pipelines*” on page 867. The following sections describe the default behaviour.

You get messages displaying the state of the stages when this occurs. A snapshot of the pipeline control blocks can be written to disk. This snapshot is provided as a service aid; the format is “undefined”. On CMS, the snapshot is appended to the file *PIPDUMP LISTING* unless the global variable *PIPDUMP* is set to *OFF*. (Use *GLOBALV* to set global variables.)

Also see “Wondering If It Is a Bug?” on page 134.

Chapter 10. Pipeline Idioms—or—Frequently Asked Questions

This chapter contains the answers to question that are often asked on the online *CMS Pipelines* fora. The intent of this chapter is first of all to point you to the built-in programs you need to investigate for some particular function; once you have an idea which function to use, refer to the authoritative description in Chapter 23, “Inventory of Built-in Programs” on page 253. But we also hope you will take note of the examples of *pipethink* that we show.

How Can I Do xxx and Get the Result into REXX Variables?

This question is probably the first one you will ask when you start to use *CMS Pipelines*.

For example, to store the file names, types, and modes of all files on all modes that are accessed read/write into a list of REXX variables starting with FILE.1 and setting FILE.0 to the number of items stored in the “stemmed array”:

Figure 226. Loading a Stemmed Array

```

/* Find all files */
'PIPE',                               /* The CMS command          */
'|command QUERY ACCESSED R/W',        /* Find accessed modes     */
'|drop 1',                             /* Drop the title          */
'|spec /LISTFILE * * / 1 1 next',     /* Build listfile command  */
'|command',                             /* Issue listfile commands */
'|stem file.'                          /* Store file names in array */

```

The solution shows some amount of *pipethink* by running one CMS command and processing the response to generate a list of other CMS commands, which are then issued. Note the filtering of the response from the first command: the heading line must be discarded.

The solution above is a good starting point for developing your business application. But do not stop here; read on! The asking of the question *per se* also deserves comment.

- Always do as much in the pipeline as you can once you have your data in there; it is much more efficient to bring *CMS Pipelines* built-in programs to bear on your data in the pipeline than to write the same function as REXX procedural code. And it is also a lot fewer keystrokes, because the *CMS Pipelines* notation is much more compact than the corresponding procedural code.
- Moving function from procedural REXX to functional *CMS Pipelines* pipeline specifications will make both you and your system more productive.
- If you are enhancing an existing program to use *CMS Pipelines* for I/O, look further in the program and you will most likely see a loop over the stem just loaded with data. Are you sure this loop could not be performed with *CMS Pipelines* built-in programs?

Knowing how to apply *CMS Pipelines* built-in programs is the hallmark of a skilled pipeline programmer. Once you can bring the full power of *CMS Pipelines* to bear on your data while they are in the pipeline, you can call yourself a *journeyman plumber*.

Locating One of Several Targets

Use *all* to select a record that contains one of several strings. Use the *zone* control to limit the search to a particular input range.

```
... | zone 10-16 all /BDLVMA/ ! /BDLVMP/ | ...
```

But *all* is more. It supports an expression of targets which are separated by OR operators and AND operators, using the normal precedence that AND groups its operands closer than OR. So an expression like

```
/a/ ! /b/ & /c/
```

is evaluated as if it were coded as

```
/a/ ! ( /b/ & /c/ )
```

Use parentheses to specify a different grouping. The NOT operator negates a target; the line is selected if it does not contain the string. Finally, the target can be specified as a hexadecimal or binary literal.

```
... | all (/a/ ! /b/) & ~ x02 | ...
```

In this example, records that are selected do not contain X'02', but do contain either "a" or "b".

Making Things Case Insensitive

All built-in programs, for which it makes a difference, support options to make them work without making distinctions between upper case and lower case letters (at least as long as you are content with "letter" meaning one of the English alphabet's twenty-six letters).

- *change* ANYCASE performs caseless substitutions; it even goes the extra mile to try to maintain the case of the substituted text.
- Other built-in programs support the ANYCASE keyword to make their processing independent of the character case. Beware, however, that specifying ANYCASE may degrade performance of the program significantly. In particular, *sort* is optimised when ANYCASE is omitted.
- User written selection stage that do not provide an option for caseless selection can run under *casei* to work without regard to the case of the data being processed.

If the default upper case translation is not appropriate or the function you wish to perform is not listed above (or you wish to optimise performance for a large file), you must resort to brute force: Prefix the record with a temporary field that contains the data in upper case; perform the operation you wish to perform, using this temporary field; and delete the field when you are done:

```
... | spec 1.3 1 1-* 4 | xlate 1.3 upper | sort 1.3 | spec 4-* | ...
```

See also "Destructive Testing" on page 84.

Numeric Sorting

The *sort* built-in program sorts by the binary contents of the key field. You can use the PAD option to extend shorter keys with a pad character on the right for purposes of comparison, but for sorting numbers you need to pad on the left and *sort* does not support this.

As always, when a built-in program does almost what you want, but not quite, put on your *pipethink* cap and create a sort key that is aligned to the right. *spec* does that easily enough.

If the numbers are unsigned integers and you know an upper limit to the field length, you can use the approach in Figure 227:

<i>Figure 227. NS0 EXEC—Naive Numeric Sort of Unsigned Numbers</i>	
<pre> /* Numerical sort of unsigned number in word2 of record */ Signal on novalue Address COMMAND 'PIPE (name NS0)', ' literal Donna 150,000', /* some random numbers */ ' literal Poul 50,000', ' literal Bob 100,000', ' spec word 2 1.15 right', /* Align sort key */ '1-* next', /* and the entire record */ ' sort 1.15', /* Sort on the sort key */ ' spec 16-* 1', /* Discard key */ ' console' Exit RC </pre>	
<pre> ns0 ▶Poul 50,000 ▶Bob 100,000 ▶Donna 150,000 ▶Ready; </pre>	

The sort key is put in the first fifteen positions of the record, aligned to the right, before the sort; it is removed after the sort.

Putting the sort key first in the record as a field that has fixed length has also the advantage of improving *sort*'s performance.

A sort key is definitely required when the data can contain both positive and negative numbers or even decimal fractions. One of the tricks to sort a mixture of negative and positive numbers is to add (or subtract) some very large number to all the keys so that the resulting number is positive for all key values. The example in Figure 228 on page 130 shows a revision of the previous example:

Figure 228. NSI EXEC—Sort Fractional Signed Decimal Numbers

```

/* Numerical sort of signed number */
Signal on novalue
Address COMMAND
'PIPE (name NS1)',
'|literal 2,735.8 7.99 -15 -3,586.99', /* Some random numbers */
'|literal 0.0001 0 -0.0001', /* Not random numbers */
'|split', /* Make them records */
'|spec word 1 1 1-* nextword', /* Make copy of number first */
'|change word 1 /,/', /* Remove commas in number */
'|spec a: w1 .', /* Convert first word to counter */
'|print a+1000000000 pic 999999999v9999 1', /* Make aligned */
'|fieldseparator blank field 2-* nextword', /* Rest of record */
'|sort 1.15',
'|console'

Exit RC

ns1
▶09999964130100 -3,586.99
▶0999999850000 -15
▶0999999999999 -0.0001
▶1000000000000 0
▶1000000000001 0.0001
▶1000000079900 7.99
▶1000027358000 2,735.8
▶Ready;

```

In the example above, the output record contains the sort key in the first column and the original number in the remainder of the record. The number was converted to excess-1000000000 notation by adding this huge number to all keys. You can see that for values that are zero or positive the key starts with “1”, whereas the key for records with a negative values starts with “0”.

There were several tricks to generating this sort key:

- The commas for thousands were removed by *change*. You have to be more careful about the numbers when you use *spec*'s numerical capabilities. On the other hand, the numbers could be in any format that *spec* supports.
- The remainder of the record after the first word is referenced as a field rather than as a word. This retains leading and trailing blanks.
- The picture that specifies the format of the output number must be large enough to accommodate the numerically largest key as well as the largest number of decimals you consider significant. You can use up to thirty digit selectors; note that one must be reserved for the initial digit that distinguishes between positive and negative numbers.
- The decimal point is elided in the sort key; its implied position is indicated by the “v”.

Internally, *spec* uses a floating point decimal format that has thirty-one digits precision and (for practical uses) an infinite range of exponents. You can use all of *spec*'s facilities to round, truncate, and so on. The number you add can be any number as long as it is equal to or larger than the negative of the smallest key.

If the input numbers are in the Continental European notation (periods for the thousands; comma for the decimal point), you can delete the periods with *change* and then use *xlate* to turn the comma into a period.

! Hexadecimal Sorting

! When sorting values in hexadecimal notation, a simple *sort* stage does not work because
! the letters “A” to “F” should come after the digits “0” to “9”. Like with sorting numeric
! values it would be possible to use the X2D conversion of *spec* to create a temporary sort
! key and remove that again after sorting.

! Instead of converting the value to decimal notation, you can use *xlate* to translate the 6
! letters to code points X'FA' to X'FF' and use *sort* on the records.

! To avoid adding the temporary sort key to the records, experienced plumbers swap the two
! pairs of code points with *xlate* like this.

```
! xlate *-* A-F FA-FF FA-FF A-F
```

! After sorting the records, the original contents of the records is restored with *xlate* again to
! swap the code points back.

! The two *xlate* stages and the *sort* can be wrapped in a simple REXX filter.

```
! /* HEXSORT REXX      Sort hex numbers      */
```

```
! parse arg parms  
! hex = 'A-F FA-FF FA-FF A-F'
```

```
! 'addpipe (name HEXSORT.REXX:5)',  
!   | *.input: ',  
!   | xlate *-*' hex,  
!   | sort' parms,  
!   | xlate *-*' hex,  
!   | *.output:'
```

```
! return rc
```

! A subroutine like this can be used just like *sort* would be used, including for example with
! options like ANYCASE, COUNT and UNIQUE.

Obtaining the Length of Records

count can be used to find the length of the shortest and longest record in a file.

To prefix each record with a halfword length field in binary:

```
... | addrdw cms | ...
```

The records must be shorter than 64K. Use the keyword CMS4 to prefix four bytes binary length.

addrdw was added in *CMS Pipelines* level 1.1.9. You might see an earlier idiom to achieve this:

```
... | spec 1-* v2c 1 | pad 2 00 | ...
```

The *pad* stage compensates for *spec* considering a null record to contain no data, rather than containing a null field. Use an additional *spec* stage to convert the binary halfword to decimal:

Pipeline Idioms

```
... | spec 1.2 c2d 1 3-* nextword | ...
```

Running a Filter on Part of the Record

Several of the built-in programs allow a range specification, for example, `locate 30.3 /abc/`, which selects records that have “abc” in columns 30-32. Some of these filters, such as `locate`, `xlater`, and `pick`, allow that range to be a column range, a word range, or a field range. They also allow negative ranges, which are counted from the end of the records. Refer to `inputRange` for the complete description.

`zone` can also be used to run a selection stage against a range (column, word, or field).

The general solution is to split the record into three pieces; apply the operation; and join the parts back together. This approach will work as long as the operation does not delay the record:

Figure 229. Applying a Filter to Part of the Record

```
'callpipe (end ?)',
  '?*:', /* Input records come here */
  '|c1: chop 17', /* Take first 17 columns */
  '|i: faninany', /* Merge together again */
  '| join 2', /* Make one record of the 3 pieces */
  '|*:', /* Write to output */
  '?c1:', /* Columns 18-* */
  '|c2: chop 7', /* Take seven */
  '| yourpgm', /* Do your thing */
  '|i:', /* Pass on to merge */
  '?c2:', /* Pass 25-* */
  '|i:'
```

If `chop` is not appropriate to select the part of the record you wish, you must turn to `spec` (or a REXX filter of your own):

Figure 230. Applying a Filter to Part of the Record, Differently

```
'callpipe (end ?)',
  '?*:', /* Input records come here */
  '|o: fanout', /* Make copies */
  '| spec word 1.2', /* Take two words */
  '|i: faninany ', /* Merge together again */
  '| join 2 / /', /* Make one record of the 3 pieces */
  '|*:', /* Write to output */
  '?o:', /* The whole record here too */
  '| spec word 3', /* Take the third word */
  '| yourpgm', /* Do your thing */
  '|i:', /* Pass on to merge */
  '?o:', /* The whole record here too */
  '| spec word 4-*', /* Take the remainder */
  '|i:'
```

This solution is not as robust as the previous one, because multiple blanks surrounding the third word are lost. Splitting the record with `threeway` may prove to be more robust.

When the Sort Does Not

When *sort* does not appear to sort properly, it may be that it properly sorts some other columns than the ones you thought you were sorting. Be sure not to lapse into the syntax of the *CMS* or *XEDIT* sort commands; they specify ranges differently than the *CMS Pipelines* sort does. Note the difference between these:

```
... | sort 1 8 | ...
... | sort 1-8 | ...
```

The first one sorts on the first column and then on the eighth column, ignoring the contents of columns two to seven. The second one sorts on the first eight columns. If you are sorting records that contain user IDs, chances are that you want the second one.

Why Does QUERY CMSTYPE not Work?

In the program below, the variable *cmstype* will always have the value *RT*.

```
pipe cms query cmstype | spec w3 | var cmstype
```

The reason is that the *cms* and *command* host command interfaces each maintain a separate *CMSTYPE* flag and set it to *RT* when they start. Use this subterfuge to obtain the *CMSTYPE* setting that applies outside the pipeline:

```
pipe subcom cms query cmstype ( lifo | hole | append stack | take 1 | ...
```

You can indeed change the flag while the host command interface is running, even by sending it commands to that effect, but the flag is discarded when the stage terminates. Thus, you cannot affect a permanent change to the *CMSTYPE* flag through the *cms* and *command* host command interfaces.

Why Does SPLIT 80 Not Work?

split 80 does work; it splits the records at each *X'80'* character.

If you wish to chop long records into 80-byte chunks, you should investigate one of these approaches:

- *deblock 80*. This produces as many records as needed to write all data in each input record into output records that are at most eighty bytes long. An input record that is shorter than eighty bytes is passed unchanged. *deblock 80* respects input record boundaries; if the length of the input record is not a multiple of eighty, the last record of the batch produced from an input record will be a short record. *deblock 80* is probably what you need.
- *fblock 80*. This ignores input record boundaries; that is, it treats the input data as a byte stream. This byte stream is then written eighty bytes at a time. Thus, an output record can contain data from adjacent input records; the only possible short output record is the last one.
- *spec 1-80 1 WRITE 81-* NEXT*. This writes two records for each input record. The first output record contains the first eighty bytes of the input record (or the entire input record if it is shorter than eighty bytes). The balance of the input record is passed in the second output record (or it is null). You can use *locate 1* to discard null records.
- *chop 80*. This produces the first eighty bytes on the primary output stream and the remainder of the record (or a null one) on the secondary output stream. You can use *faninany* to merge the two parts of the original record into sequence:

Pipeline Idioms

```
'PIPE (end ? name PIPFAQ.SCRIPT:244)',  
  '?...',  
  '| c: chop 80',  
  '| i: faninany',  
  '| ...',  
  '? c:',  
  '| i:'
```

Why Can't I Update a Stemmed Array?

You can.

If you use *stem* to read and write the same stemmed array in a pipeline, you must be sure that writing back to the array does not overtake reading from the array. If it does, you will not get the result you expect (or at least the one we expect you will expect), because you might have a “destructive overlap”.

Be sure to buffer the array before writing it back to the stem when your process can produce more records than it reads:

Figure 231. Updating a Stemmed Array

```
'PIPE stem x. | split * | buffer | stem x.'
```

However, no points can be awarded for the artistic impression. Surely you can move more of the processing of the stem into the pipeline to take advantage of the speed of built-in filters relative to interpreted REXX; it is likely that you can do away with the stem entirely.

Wondering If It Is a Bug?

If something “does not work”, it could be an error in *CMS Pipelines*. Though they have been observed occasionally, errors in *CMS Pipelines* are rare. It is more likely that your expectations of how *CMS Pipelines* works are different from how it was designed to work. But documentation is just as important a part of a product as is the code; if you read the documentation and it led you to believe that the code should work differently than it does, the documentation must be improved. Submit a Reader Comment Form to let us know where we failed.

If it just “does not work”, the reason might be one of these:

- If no data come out of the pipeline, it might be because no data came into it. Maybe an intermediate stage has terminated for some reason. Use *count* with a secondary output stream in front of a stage to count the amount of data that are consumed by the stage. Normally, you would count the number of lines that are actually read by a stage.
- If this is your first attempt at writing a multistream pipeline, you might easily be tripped by the way the topology of a pipeline network is specified. Check your pipeline against one of the examples in Chapter 5, “Using Multistream Pipelines” on page 74. If you are using a multistream program that should be in the middle of the pipeline, ensure that there are no end characters adjacent to the label reference to it.
- If the trouble is with *lookup*, it could be that the secondary and tertiary streams are connected incorrectly. It is very tempting to leave the secondary output stream unconnected in one’s first pipeline that uses *lookup*.

If you are still convinced the problem is in *CMS Pipelines*, try to gather as much information as possible before reporting the suspected bug:

- Query the level of *CMS Pipelines* are you using. (PIPE QUERY LEVEL will tell you this.) Which operating system is it running under and what is the release of this operating system?
- Try to reduce the pipeline that produces the failure to a few simple stages using little data. With a simple test case we are able to understand the problem faster and thus provide you with a fix in a timely manner.
- Show the exact pipeline specification that causes the failure. If you cannot embed an EXEC, use cut and paste to be sure you get the exact command you typed.
- Show your input data and the output result you obtained. Also show the output result you expected or indicate where the actual is different from your expectations.

Pipeline Idioms

Part 3. Specialised Topics, Tutorials

This part of the book contains tutorials and chapters on specialised topics, some of which may not be of interest in your installation.

Chapter 11, “Accessing and Maintaining Relational Databases (DB2 Tables)” on page 138 explains how to access relational databases and process the result of a query.

Chapter 12, “Using *CMS Pipelines* with Interactive System Productivity Facility” on page 145 explains how to access Interactive Systems Productivity Facility to get or set function pool variables and lines in tables.

Chapter 13, “SPOOL Files and Virtual SPOOL Devices on VM” on page 149 explains about unit record equipment and SPOOL files on z/VM.

Chapter 14, “Using VMCF with *CMS Pipelines*” on page 156 describes how to build a VMCF server.

Chapter 15, “Event-driven Pipelines in Clients and Servers” on page 160 explains about using *CMS Pipelines* in client/server applications.

Chapter 16, “*spec* Tutorial” on page 166 explains the workings of *spec* in easy steps.

Chapter 17, “Rita, the *CMS Pipelines* Runtime Profiler” on page 208 introduces the *CMS Pipelines* performance tool.

Chapter 18, “Using VM Data Spaces with *CMS Pipelines*” on page 210 describes how to use data spaces with *CMS Pipelines*.

Chapter 19, “*CMS Pipelines* Built-in Programs supporting Data Spaces” on page 220 introduces built-in programs that support ALET operands.

Chapter 11. Accessing and Maintaining Relational Databases (DB2 Tables)

The *sql* device driver interfaces *CMS Pipelines* and the relational database products:

- IBM DB2 Server for VSE and VM.
- IBM Database 2 for z/OS.

Collectively these products are referred to as DB2. *sql* processes statements in the Structured Query Language (SQL).

Potential users of *sql* include:

- Users of DB2 who are new to *CMS Pipelines*. *sqlselect* is a sample program to format a query much as ISQL does it. Input lines read by *sql* are issued as SQL statements, or are data for INSERT statements; the result of queries is written to the output. The format of data is the internal DB2 format (see *DB2 Server for VSE & VM Application Programming*); use *spec* to convert between external and internal formats.
- *CMS Pipelines* users who are new to DB2. Basic DB2 education is outside the scope of this book; refer to *DB2 Server for VSE & VM Application Programming*.

Several tasks are performed before a *CMS Pipelines* user can issue SQL statements through *sql*:

- DB2 must know about *CMS Pipelines*. On CMS, this process is called *preparing the access module*. It is performed once by your system support staff. On z/OS, it is called *binding the plan*. Help for *sql* as well as *CMS Pipelines* installation procedures describe this process.
- If you are going to use Distributed Relational Database Access (DRDA), you must ensure that all other systems know about *CMS Pipelines*. Unload the plan from the system where you have installed *CMS Pipelines* and bind it at the other systems.
- You must be registered as a DB2 user. Contact your database administrator if you are not already registered. Your installation may have granted everyone connect authority; you can query tables once you have connect authority.
- To create tables, you must have a DBSPACE or write privileges to a space owned by someone else. Your database administrator allocates a space to you.
- On CMS, you must issue the command SQLINIT before you can access SQL tables; this establishes the connection to the database server. On z/OS, the option SUBSYSID specifies which subsystem you wish to connect to, if it is different from the default for your installation.

The following description is slanted towards CMS. z/OS users should substitute “DB2 subsystem” for “DB2 server”.

sqlselect—Format a Query

CMS Pipelines provides *sqlselect*, which formats a query for presentation on the terminal. The filter takes a query as the argument, describes the query, and formats the result; see Figure 232 on page 139. 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.

Figure 232. SQLSELECT Examples

```

pipe sqlselect project_name from sqldb.projects | console
▶PROJECT_NAME---
▶BLUE MACHINE
▶GREEN MACHINE
▶ORANGE MACHINE
▶RED MACHINE
▶WHITE MACHINE
▶Ready;

pipe sql describe select salary, name from q.staff | console
▶485      DECIMAL      7,2      4 SALARY
▶449      VARCHAR      9        11 NAME
▶Ready;

pipe sqlselect salary, name from q.staff where years is null | console
▶SALARY--- NAME-----
▶+16808.30 QUIGLEY
▶+13504.60 JAMES
▶+12954.75 NAUGHTON
▶+11508.60 SCOUTTEN
▶Ready;

```

Creating, Loading, and Querying a Table

Use *sql* to query and maintain DB2 tables.

Two ways to create a table are shown in Figure 233. The first example shows how to issue a single SQL statement; the second example shows that *sql EXECUTE* reads statements from its primary input stream. The point is that you can supply many SQL statements to a single invocation of the *sql* device driver.

Figure 233. Creating a Sample Table

```

pipe sql execute create table jtest (kwd char(8), text varchar(80))
▶Ready;

pipe literal create table jtest (kwd char(8), text varchar(80))|sql execute
▶Ready;

```

Use *sql INSERT* to load data in the table (see Figure 234 on page 140). The first eight characters of each record are stored in the column *kwd*; the remainder of the record is loaded into the column *text*.

To insert the values, build complete insert statements using literal data:

Figure 234. Inserting Rows in a Table on z/OS

```

/* Insert lines in a table */
signal on novalue
address Attach
'PIPE',
  'literal DMS    Conversational Monitor System|',
  'literal HCP    Control Program|',
  'literal DMT    Remote Spooling Communication System|',
'spec /insert into jtest (kwd, text) values("/ 1',
  '1.8 next  /", "/ next  9-* next  /")/ next',
'sql execute'
exit RC

```

Note that you must enter the names of the columns, even when you are setting all of them. On z/OS, you will get a strange SQLCODE if you omit the column names.

On CMS you can use a faster underlying interface (inserting on a cursor) by omitting the value clause and supplying the values for all columns in the appropriate format. *spec* is used with a conversion option to generate the halfword length required for the variable character string:

Figure 235. Inserting Rows in a Table on CMS

```

/* Insert lines in a table */
signal on novalue
address command
'PIPE',
  'literal DMS    Conversational Monitor System|',
  'literal HCP    Control Program|',
  'literal DMT    Remote Spooling Communication System|',
'spec 1.8 1 9-* v2c 9|',
'sql insert into jtest'
exit RC

```

All columns defined for the table are loaded with data from the input record when *sql* INSERT is used without further operands. *sql* obtains the length of each column from DB2 Server for VM; data loaded must be in the format used by DB2 Server for VM, which in general involves conversion. The *spec* stage copies the first eight characters of each record without change; it then inserts a halfword field with the number of bytes remaining in the input record and copies the rest of the input record after this halfword. This is the format required by DB2 Server for VM for a row with a fixed and a varying length character variable.

Figure 236 shows how to use *sql* DESCRIBE SELECT to see the format of the input record or the result of a query.

Figure 236. Describing a Query

```

pipe sql describe select * from jtest | console
▶453    CHAR          8      8 KWD
▶449    VARCHAR       80     82 TEXT
▶Ready;

```


Each line describes a column in the table. The first column of the record is the numeric SQL field code. It is decoded in the next column. A column with the length (or precision) of the field as perceived by DB2 is next. The following number 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.

sql SELECT queries a table (see Figure 237).

Figure 237. Querying a Table

```
pipe sql select * from jtest | console
▶""DMT      """"Remote Spooling Communication System
▶""DMK      """"Control Program
▶""DMS      """"Conversational Monitor System
▶Ready;
```

The double quotes in Figure 237 represent unprintable binary data. The first two positions of each column contain the indicator word that specifies whether the column is null or contains data. This information may be required to process the result of a query of a table that contains columns that can contain the null value (no data). Figure 238 shows how indicator words are suppressed in the output record; the query seen by DB2 is the same in both cases.

Figure 238. Querying a Table, Suppressing Indicator Words

```
pipe sql noindicators select * from jtest | console
▶DMT      ""Remote Spooling Communication System
▶DMK      ""Control Program
▶DMS      ""Conversational Monitor System
▶Ready;
```

The remaining two unprintable bytes contain the length, in binary, of the varying field. Use *spec* to discard these columns. As an alternative, Figure 239 shows how to use *spec* to format binary data.

Figure 239. Querying a Table, Formatting Field Length

```
/* Query the test table without formatting */
Signal on novalue
Address command 'PIPE',
  'sql noindicators select * from jtest |',
  'spec 1.3 1 9.2 c2d 5.2 right 11-* 8 |',
  'console'
Exit RC

sqlq3
▶DMT 36 Remote Spooling Communication System
▶DMK 15 Control Program
▶DMS 29 Conversational Monitor System
▶Ready;
```

spec supports conversion between character and binary or floating point, as well as constructing varying length character fields.

Using DB2

In Figure 240 on page 142, *sqlselect* formats a query against the sample table.

```
Figure 240. Another SQLSELECT Sample

pipe sqlselect * from jtest | console
▶KWD----- TEXT-----
▶DMT      Remote Spooling Communication System
▶DMK      Control Program
▶DMS      Conversational Monitor System
▶Ready;
```

Using *spec* to Convert Fields

Input and output records from *sql* have data in the format that is defined for the table. For instance, when a column is specified as *SMALLINT*, the corresponding field in a record is a two-byte binary integer.

Use *spec* to convert from readable formats to the internal ones. The sample program *sqlselect* shows how to format DB2 data on output. Figure 241 shows how to convert some DB2 data types. The input record is assumed to contain a single field.

Data Type	Conversion to Internal Format
Character string that has a fixed length.	This example pads or truncates the field to eight bytes. spec 1-* 1.8
Character string that has a varying length.	spec 1-* v2c 1
Large integer	spec 1-* d2c 1
Small integer	spec 1-* d2c 1.2 right
Floating point	spec 1-* f2c 1
Decimal	If the number should contain two decimals: spec 1-* p2c(2) 1

About 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 CMS reaching end of command. Unless instructed by an option, *sql* performs an explicit commit and relinquishes the connection to the database virtual machine when processing is complete. Use the option *COMMIT* when you wish the unit of work to be committed without releasing the connection to the database machine. 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 CMS or pipeline error condition.

Using Multiple Streams with *sql* Stages

sql EXECUTE processes multiple input and output streams when the primary input stream has multiple insert or query statements, or a mixture of these. Each insert statement causes *sql* to read records from a separate input stream, starting with stream number 1; there must be as many additional input streams defined as there are insert statements.

The result of the first query is written to the primary output stream. If the secondary output stream is defined and connected, the result of the second query is written there, and so on. More queries are allowed than there are streams defined. The output records from the last queries are written to the highest numbered stream defined.

Using Concurrent *sql* Stages

You can process the results of a query to construct SQL 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 multiple cursors.

The option NOCOMMIT must be specified when multiple *sql* stages are running concurrently. Each stage uses its own cursor; the module is prepared for up to ten cursors.

If one of the stages fails with a DB2 error, the unit of work is rolled back and all other *sql* stages fail if they access DB2 after the error occurred. Use a buffer stage to isolate the programs when building SQL statements from the result of a query. This ensures that the initial query is complete before a subsequent stage starts processing. You can also process the 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 completes OK.

CMS Considerations

Obtaining Help

DB2 Server for VM stores help information in tables. If you have connect privileges and have run SQLINIT, you can use *help* SQL to access these tables. Specify the topic about which you wish help as the argument. This may be an SQL statement or a numeric return code. Use *help* SQLCODE to obtain help for the last return code received from SQL; *help* SQLCODE 1 displays help for the second last return code received, and so on.

Because CMS HELP has no interface to receive the information to display, it is displayed in an XEDIT session.

Figure 242. Sample Pipe Commands to Obtain Help for DB2

```
pipe help sql select
pipe help sql 105
pipe help sqlcodes
```

Figure 243 on page 144 shows a session where a user accesses DB2 for the first time. The first attempt to obtain help fails. Simply issuing the SQLINIT command does not help because the EXEC is not available. Having linked and accessed (minidisks and mode letters may be different in your installation), the user runs the initialisation procedure and obtains help.

Using DB2

Figure 243. Running SQLINIT

```
pipe help sql select
▶SQL RC -934: Unable to find module ARISRMBT; run SQLINIT.
▶... Issued from stage 1 of pipeline 1.
▶... Running "sql nocommit select item from sqldb.systext1 wher".
▶Unable to obtain help from SQL (return code -934).
▶... Issued from stage 1 of pipeline 1.
▶... Running "help sql select".
▶Ready(00364); T=0.09/0.12 13:19:43

sqlinit
▶Unknown CP/CMS command

link sqldb 195 195 rr
▶Ready; T=0.01/0.01 13:20:16

acc 195 t
▶T (195) R/0
▶Ready; T=0.01/0.01 13:20:23

sqlinit db(sqldb)
▶Ready; T=0.09/0.14 13:20:39

pipe help sql select
▶Ready; T=0.22/0.37 13:21:26
```

help SQL uses *sql*; Figure 244 shows the response when the access module has not been generated by your systems support staff.

Figure 244. When the SQL Access Module Is not Generated

```
pipe help sql select
▶SQL RC -805: Access module 5785RAC .PIPSQI not found; refer to help for SQL to generate acc
▶... Issued from stage 1 of pipeline 1.
▶... Running "sql nocommit select item from sqldb.systext1 wher".
▶Unable to obtain help from SQL (return code -805).
▶... Issued from stage 1 of pipeline 1.
▶... Running "help sql select".
▶Ready(00364); T=0.10/0.15 16:51:50
```

The example in Figure 244 was run in the PIP style. If the access modules are generated for the DMS style, it may help to use this style instead.

Chapter 12. Using CMS Pipelines with Interactive System Productivity Facility

Interactive System Productivity Facility (ISPF) maintains (among many other things) tables and pools of variables. ISPF services build lines in tables based on the contents of *function pool variables*.

CMS Pipelines users can issue ISPF requests in two ways:

- Any ISPF request that can be processed by ISPEXEC can also be issued by passing a record containing the request to *subcom* ISPEXEC. This is a consequence of the way subcommand environments work.
- The built-in program *ispf* allows access to function pool variables and to tables. It can copy function pool variables into a record, which is then written into the pipeline; or it can replace the contents of function pool variables with data read from the pipeline. This can be combined with some of the ISPF table service requests.

ISPF is also a dialog manager. If you normally work within an ISPF dialog, you can define a PIPE command to ISPF, as described in “Defining PIPE to ISPF” on page 148.

Issuing ISPF Commands from REXX Filters

On CMS, the REXX Address instruction can reach any defined environment; this is often the most convenient way to issue ISPF service requests. When issuing a sequence of ISPF service requests several times, it may, however, be easier to store the commands in a stemmed array and pass it to *subcom* ISPEXEC to be processed.

On z/OS, there is no choice: *subcom* ISPEXEC is the only way to issue ISPF service requests from a REXX filter because a REXX filter executes in a reentrant environment, which is not merged with TSO and therefore has no ISPEXEC environment defined. On the other hand, *subcom* selects subcommand environments in the default REXX environment.

Accessing ISPF Tables

You must create, open, and close tables using standard ISPF commands (which you can issue through *subcom* ISPEXEC).

Figure 245. Creating and Opening an ISPF Table

```
'PIPE (name PIPUISPF)',
  '|literal tcreate files names(ddname dsname) write replace',
  '|subcom ispexec'
```

Once a table is open you can read rows from it and you can add or replace rows in it. As an example, Figure 246 on page 146 shows how to create a table that contains a row for each allocated DDNAME. It is assumed that the table is defined to contain the variables DDNAME and DSNAME.

Figure 246. Building a Table

```

/* Load LISTA output into table */
address link,
'PIPE (name PIPUISPF)',
  '|tso lista status',          /* Issue TSO command      */
  '|drop 1',                   /* Drop heading           */
  '|nfind TERM',              /* Discard TERMFILE      */
  '|nfind NULL',              /* Discard dummy allocations */
  '|spec 1-* 12 read 3.10 1', /* Splice lines          */
  '|nfind _',                  /* Discard catenations   */
  '|ispf tbadd files ddname 1.8 dsname 12.44' /* Build table */

```

In this example, the stages up to the last one transform the response to the TSO query into a file that has one line for each DDNAME that is allocated to a real data set. *ispf* TBADD then performs these steps for each input record:

- It peeks at the input record without consuming it.
- It issues a VREPLACE service request to set the variable DDNAME to the contents of columns 3 through 10 and the variable DSNAME to the contents of the record from column 12 onward.
- It issues a TBADD service request to copy the contents of the function pool variables into a row of the table. Note that the table can have more columns than the two that are specified as the argument. The additional columns would be set from the current contents of the respective variables (which would be specified when the table was defined).
- It copies the input record to the primary output stream. In this example, the output stream is not connected; the output record is discarded.
- It consumes the input record.

Only one column of the table is read into the pipeline in the example below:

Figure 247. Reading an ISPF Table into the Pipeline

```

/* Read table into pipeline */
address ispxec 'tbttop files' /* Move to top */
address link,
'PIPE (name PIPUISPF)',
  '|ispf tbskip files ddname 1.8',
  '|...'

```

ispf TBSKIP performs these steps to build each output record:

- Call the TBSKIP service to move to the next row of the table and set the values in the function pool.
- Call the VCOPY service to copy the contents of the variable DDNAME into the stage's output buffer.
- Write the output record into the pipeline.

This process continues until ISPF sets a return code to indicate that the end of the table has been reached.

Note that ISPF sets a function pool variable for each column in the table even though the example above copies only one variable into the pipeline. The remaining variables remain in the function pool where they can be used by other requests, for instance to update a table:

Figure 248. Updating a Column of an ISPF Table

```

/* Modify table */
address ispexec 'tbttop files'          /* Move to top          */
address link
'PIPE (name PIPUISPF)',
'|ispf tbskip files ddname 1.8',
'|xlate lower',
'|ispf tbput files ddname 1.8'

```

These steps are performed for each row of the table:

- The first stage calls the TBSKIP service to move to the next row of the table and set the values in the function pool.
- It then calls the VCOPY service to copy the contents of the variable DDNAME into the stage's output buffer.
- It then writes the output record into the pipeline and waits for the write to complete.
- *xlate* peeks at the input record and builds a record containing the lower case DDNAME in an output buffer.
- It then writes this record to the pipeline and waits for the write to complete.
- The third stage peeks at the input record and then issues a call to the VREPLACE service to replace the contents of the variable DDNAME.
- It then issues a call to the TBPUT service to copy the contents of the function pool variables into the current row of the table. Note that this replaces all columns of the row; the other columns were copied into the function pool at the first stage and have remained there unmodified.
- Finally the third stage consumes the input record.
- This causes the *xlate* stage to consume its input record.
- The first stage is now ready to perform another cycle.

xlate does not delay the record. It is important that no stage delays the record between the one that reads a line from the table up to the one that replaces the line in the table. If the data were to be delayed, the wrong line in the table might be updated.

Accessing ISPF Function Pool Variables

When the “canned” functions to access ISPF tables (TBSKIP, TBADD, TBMOD, TBPUT) do not perform the function you require, you can use the VCOPY and VREPLACE options on *ispf* to access variables directly without reference to tables and you can issue table service requests directly to ISPF.

On TSO, the function pool can never be the variable pool of a REXX filter; on CMS it might be. To import values from the ISPF function pool into the variable pool of a REXX filter:

Figure 249. Importing and Exporting ISPF Variables

```

/* Import "myvar" from ISPF */
'callpipe literal | ispf vcopy myvar 1.20 | var myvar'
myvar=time() myvar
/* Export it back again */
'callpipe var myvar | ispf vreplace myvar 1-*'

```

Note that *ispf* VCOPY needs an input record to trigger a cycle; without the input record it would produce no output.

Interaction (on TSO) Between ISPF and Stages that Access REXX Variables

The function pool that ISPF maintains for REXX variables is in the REXX environment that ISPF creates when it initialises. On the other hand, REXX filters run in separate reentrant environments which each contain their own variable pools. Thus, *ispf* may be accessing a different variable pool than does, for example, *var*.

Defining PIPE to ISPF

If you would like to be able to issue the PIPE command directly from an ISPF command line without using a TSO prefix, you must first define the PIPE command to ISPF.

You can do this by adding a line to the table ISPCMDS, which defines the ISPF commands. You must define the PIPE command as a line mode command to ensure that ISPF refreshes the screen when the pipeline is done. Figure 250 shows the ISPF variables you should set before adding the row to the ISPCMDS table.

Figure 250. Defining the PIPE Command to ISPF.

Variable Name	Variable Contents
ZCTVERB	PIPE
ZCTTRUNC	0
ZCTACT	SELECT PGM(PIPE) PARM(&ZPARM) MODE(LINE)
ZCTDESC	TSO Pipelines Command

Chapter 13. SPOOL Files and Virtual SPOOL Devices on VM

This chapter describes z/VM SPOOL files. Though there are other types of SPOOL files in a VM system, a normal SPOOL file is created by a virtual printer, by a virtual punch, or by a virtual console. Once created, the file will reside in a queue waiting to be read by a virtual machine or waiting to be transcribed to a real printer or punch. The owner of a SPOOL file can transfer the file between the various queues.

You may wonder why VM supports these strange files; here is the story. You can skip the introduction if you still remember how to program an IBM 1401.

Introduction to Unit Record Equipment

Before electronic computers, accounting tasks were done by hand or with electromechanical *accounting machines*. (And a computer was a man using paper and pencils.)

The storage medium used was a *punched card*, in which holes were punched in a twelve by eighty array: The punched card had eighty columns of twelve rows.

Master files were stored as card files and transactions were punched into cards before being processed. A typical operation would sort the transactions, collate them into the master file (which is already sorted), print invoices and update the master file, and finally remove the transactions from the master file and collate the updated master records into the new master file.

Operators attending to accounting machines performed the tasks of taking decks of cards from the stacker (where they come out) of one device and putting them into the hopper (where they are read) of another device. An operator was expected to handle stacks of 2,000 cards with his bare hands, often turning a stack upside down in the process; there would be trouble if the cards fell on the floor. CP implements the virtual card operator by transferring SPOOL files from one queue to another one.

Originally, a *card file* was something real that you could carry around with you. An able-bodied person can comfortably carry a box of 2,000 cards under each arm. Programs that were larger than 4kloc required a trolley for transportation (or several programmers).

Punched cards were not made redundant overnight by the introduction of electronic computers, however. Input was in cards well into the Nineteen-seventies. Though real cards are no longer used to store files, they are still very much in evidence in VM/CMS to support virtual reader/punch and printers.

A system running VM/370 would typically have an IBM 2540 card reader/punch and an IBM 1403 line printer. To share these devices, virtual machines were given *virtual unit record* devices. These are simulated by CP and have no real counterpart. A card file is simulated by a file in CP SPOOL; it is read by a virtual reader. A SPOOL file is created by a real reader (now extinct), a virtual punch, or a virtual printer.

In early VM days, most SPOOL files were quickly transcribed to the external medium; few users, if any, used SPOOL files for messages. The SPOOL system was not used as a repository in those days because all SPOOL files were lost if the system went down without

Unit Record Equipment

saving warm start data; that is, without being shut down properly, as would happen on a power drop.

A SPOOL file of punched cards contains records with up to 80 bytes. A column that has no holes punched is read as a blank (X'40'); you can think of short cards as padded with blanks: punching nothing leaves the column blank.

Print files contain control information in addition to character data; a printer has a *carriage* that moves the paper past the stationary printing station. When not writing text on one or more lines, the computer told the printer to skip a number of lines, or to the next page, or to the end of the page; the printer carriage then moved the paper faster than when printing. This *dual speed carriage* improved elapsed time for printing jobs, especially on sparsely printed pages.

How can a printer tell the beginning of a page? No doubt, you look for the perforation, but that was not so easy for a 1403: it had a *carriage tape*: a paper loop as long as the page or multiples thereof and about two inches wide. This paper loop was installed in the printer in a special device to read it and move it synchronised with paper movement. Each form had its own carriage tape. While printing, the printer read the carriage tape for punched holes with twelve brushes. The programmer punched holes in the carriage tape with an IBM Carriage Tape Punch and glued the ends together with IBM glue to form a band. The computer instructed the printer to *skip to a channel*, which meant until a hole was detected in the corresponding column of the carriage tape. Convention soon became that a hole in the first channel meant the top of the page. The end of the page was indicated by a hole in channel 12; it would be punched where one would print subtotals.

It was good practice to have at least one hole punched in any channel, but programmers are always too busy to remember small details like this; a printer took the skip instruction literally: when there was no hole to stop the carriage, the printer would spew out paper at high speed until the operator intervened. The original 1403 skipped heavy paper faster than it could stack it, so the paper tended to hit the lid if one skipped a long way. To avoid jams from this, the printer was normally run with the cover open: a runaway skip was quite spectacular.

With the inception of IBM System/360, control units were put between the computer and the printers as part of the standard I/O architecture which still applies. Control units are attached to channels that are programmed in a limited instruction set called *channel commands*. Each *channel command word* (abbreviated CCW) contains a command code (one byte), a buffer address, a byte count, and flag bits to control the channel. The command can be immediate with no data transfer, or it can write a line of text and then start paper movement. The command codes select the particular type of carriage movement.

On the IBM 3211 the *forms control buffer*, often abbreviated to FCB, replaced the paper carriage tape of the 1403. This electronic buffer is loaded by a channel command; data sent to the printer with the write FCB command is not the same for all printer device types.

CP SPOOL stores the CCW command code along with the data, so logically each record of a SPOOL file has a leading character which is called the *carriage control character*. It is a *machine carriage control character*, because it is the CCW command code.

We hope this long preamble explains why carriage control is important and why attention to detail is required when dealing with CP SPOOL; if you lose the carriage control character, you have lost the layout of the page, though not the words on it.

VM SPOOL Files Contain More than Just Cards

There are now four types of information associated with a SPOOL file:

- Variables such as the class of the file, the owner, and the distribution code. These are reported by the CP Query command. Because the VM/370 control block that described a SPOOL file was called an SFBLOK, this information is often referred to as the SFBLOK. z/VM does not use this control block internally, but it is still part of the diagnose interface to obtain information about a SPOOL file.
- The tag for the file, which can store up to 136 arbitrary characters. The tag is used by RSCS to specify the destination and other parameters.
- The eXtended Attribute Buffer. As far as CP is concerned, the XAB can contain up to 32K of arbitrary characters. Print Services Facility uses the XAB to store additional file attributes used for printing. The tag cannot be used, because the file might need to be transmitted by RSCS to the actual print destination.
- The data stored in the file. That is, the records that were written to the file.

A SPOOL file is created by writing records to a unit record output device; that is, a virtual punch or a virtual printer. The CLOSE command is issued when the file is complete. At this time, information is copied from the virtual device and associated with the file. This includes the SFBLOK information, the tag, and the extended attribute buffer. That is, you can change the characteristics of the device while the file is being created and the updated characteristics will be associated with the file when it is closed.

There are many attributes associated with a virtual SPOOL device. In general, a SPOOL file is created each time the device is closed; the file gets the attributes associated with the device at the time it is closed. Three attributes are particularly important when using *CMS Pipelines* unit record device drivers:

- The class. A spool file's class is a letter or a digit. A reader device can read files of a particular class only or it can read all files (class *). You can define multiple readers and use a different class with each.
- The hold status. A SPOOL file in hold cannot be read. You can create a held file or you can change a file to be held.

The hold status for a reader is interpreted differently. When the reader is NOHOLD, a file is purged after it has been read. Be sure to SPOOL the reader HOLD to retain the file.

- The continuous setting. The CLOSE command has no effect when an output device is set to be continuous. This is useful to suppress CLOSE commands in CMS commands so that you can issue your own CLOSE command, which could include the NAME operand to name the SPOOL file.

A reader that is spooled continuous reads all files of its class that are not in hold. By default, a reader reports end-of-file after each file it reads. When a reader is both held and continuous, the files are put in hold after they have been read; otherwise the reader would read the first file forever. It is unlikely that you will want to SPOOL your reader CONT.

Unit Record Equipment

Overview of Unit Record Device Drivers

CMS Pipelines provides device drivers to read and write SPOOL files as well as device drivers to control SPOOL devices. To allow you concurrent access to multiple unit record devices, a device address (or device number) can be specified with all these device drivers. The defaults are the standard devices 00C, 00D, and 00E for reader, punch, and printer, respectively.

<i>printmc</i>	Create a print file. The first column contains a machine carriage control character.
<i>punch</i>	Create a punch file. No carriage control is required, because only one operation is allowed by CP on a punch device.
<i>uro</i>	Create a print or a punch file. The first column contains a machine carriage control character.
<i>reader</i>	Read a SPOOL file. The file can be a printer or a punch file.
<i>xab</i>	Manage the eXtended Attribute Buffer of a device or a SPOOL file.

Creating a SPOOL File

Three device drivers write lines on unit record devices. Use *printmc* for a virtual printer, *punch* for a virtual punch. *uro* (unit record output) writes to either type of device.

The device address can be specified as the argument to these device drivers. The default address is 00E for *printmc* and *uro*; it is 00D for *punch*.

To give you complete control, no CP commands are issued to the virtual device: you must issue SPOOL, TAG, and CLOSE commands as required. You must also use *xab* if you wish to change the extended attribute buffer associated with a device. The SPOOL file is created by CP when you issue the CLOSE command.

Use *punch* to create a punch file. Each input line is written to the punch with X'41' carriage control, which is the only one allowed by CP except for the X'03' no operation. Note that CP truncates punch lines after column 80 without issuing a message or giving other indication of error.

Use *printmc* or *uro* to create a SPOOL file where you specify the machine carriage control as the first byte of each record. The carriage control character controls the carriage movement or the stacker selection, depending on the device type. *printmc* supports only printer devices, whereas *uro* supports both. You can find the command codes under the heading *I/O Command Codes* in the *IBM System/370 Reference Summary*, GX20-1850, and in the *IBM Enterprise System Architecture/370 Reference Summary*, GX20-0406.

Though there is only one kind of carriage control CCW codes, there are two kinds of carriage control associated with listing files: machine carriage control, which is the CCW operation code described earlier; and the more user friendly *ASA carriage control*. *CMS Pipelines* filters *mctoasa* and *asatomc* convert between the two formats. If the first position of a record with carriage control contains any of these characters it has ASA carriage control:

- 1 (X'F1') Skip to new page. The line is printed at the top of the next page. The numbers 2 through 9 and the letters A through C are defined for the other channels, but are seldom used.

- (blank) Print on the next line.
- 0 Skip one line and then print. That is, print one blank line and then the data part.
- Skip two lines before printing.
- + Overprint the line on the previous one.

printmc and *uro* require machine carriage control. Use "...|asatomc|printmc" to write a file with unknown carriage control to a printer; carriage control is converted to machine carriage control if it is not so already.

Errors on Unit Record Output Devices

When IBM System/360 was designed, standard names were given to some of the error conditions that could occur with I/O devices. One of these is "Intervention Required". It was originally meant to indicate that intervention was required by the operator; for example to clear a jam or to add paper.

CMS Pipelines reports errors on the unit record output devices with a message that includes sense data; in most cases you can ignore these hexadecimal values and concentrate on the informational message 293 that follows. (But please supply all data when you report an error.)

For all three output drivers, you will see Intervention Required if the SPOOL is full or the limit on SPOOL files is exceeded. This also occurs when you issue the CP command, NOTREADY. Issue READY to make the device ready.

Figure 251. Not Ready Punch.

```

notready d
▶Ready;

pipe literal a line | punch
▶Intervention required on 00D
▶... Issued from stage 2 of pipeline 1
▶... Running "punch"
▶Ready(00289);

ready d
▶Ready;

```

Another kind of error is to write a record with carriage control that CP does not like. Because X'41' is the only valid carriage control on a virtual punch, this one fails with X'F1':

Unit Record Equipment

Figure 252. Incorrect Stacker Select

```
pipe literal 1aline | uro d
▶FPLIOS292E I/O error on d; CSW X'039D2008 02000006', CCW X'F1600006 039
▶FPLMSG003I ... Issued from stage 2 of pipeline 1
▶FPLMSG001I ... Running "uro d"
▶FPLIOS293I Sense CmdRej
▶Ready(00292);
```

The condition is known as “Command Reject”. Ignore message 292 when message 293 shows a decoded value.

As we shall soon see, the output from *reader* has carriage control in the first position of each output record; this can be fed directly to *uro* to copy a spool file, but the virtual output device must match the type of spool file; you get the command reject error if the device and the file are incompatible.

Controlling a Unit Record Output Device

SPOOL files are used only for information interchange these days. In addition to the actual data in a file, you must also supply a destination in the tag and you must SPOOL the device to the RSCS machine.

It is practical to concentrate this in a single subroutine pipeline so that your main EXEC is not required to handle such tasks.

Figure 253. Subroutine to SPOOL and Tag a Punch

```
/* Tag and SPOOL the punch; then punch the file          */
signal on novalue
arg node user .

address command          /* Issue a number of CP and CMS commands */
'IDENTIFY(LIFO'          /* Where am I?          */
parse pull me . mynode . rscs .
spoolto=word(rscs user, 1+(node=mynode))          /* Local or remote? */
'CP SPOOL D PURGE' spoolto 'NOHOLD CLASS A'

'CP TAG DEV D' node user

address                  /* Revert to pipeline          */
'callpipe *:|punch'      /* Write the lot          */

exit RC
```

See also “Page Formatter” on page 205.

Reader SPOOL Files

SPOOL files in your virtual reader can come from several sources which have different formats. These are the general types:

Unit Record Equipment

- A virtual card punch. This is the simplest format because carriage control is limited to X'03' (no operation) and X'41', which marks data records. The longest card is 80 columns, but it can be shorter; one would normally pad them with blanks.
- A virtual printer. Such files contain data records possibly interspersed with control information (forms control buffers, etc.) The longest data record is 204 characters for a virtual 3800 file, unless the record has X'5A' carriage control (an *oversize* record with APA printer data).
- CP-generated SPOOL files, for instance a VMDUMP.
- Cardboard read by a real card reader has a format similar to a virtual punch file; such files are now almost extinct, but it is likely that the carriage control is X'42'.

How one wishes to process a file depends on the format of the file. Since SPOOL files are mostly used for electronic mail, the most common format is the virtual punch format. However, there are many protocols for the contents of a punch file with mail in it; some, such as VMSG and MAIL, are not blocked further and have a record for each line in the mail file, but other formats (notably NOTE) block the message before it is punched in cards.

Chapter 14. Using VMCF with CMS Pipelines

This chapter describes the use of *vmclient*, *vmclisten*, and *vmcdata*.

A VMCF transaction comprises sending the request from a *vmclient* (or even *vmc*) stage to a *vmclisten* stage, typically in a different virtual machine. The server uses *vmcdata* in some contexts to reject, receive, and reply.

Two similar data areas, both 40 bytes in length, are central to the workings of the VMCF stages; one of the two is present at the beginning of all records passing in and out of VMCF stages. They are documented in appendix C of *CP Programming Services* and a structure definition of each is built into *CMS Pipelines*.

VMCPARM: The VMCF parameter list is passed to CP with the VMCF diagnose, 68.

VMCMHDR: The VMCF message header is stored as part of reflecting a VMCF interrupt to a virtual machine.

For all practical purposes, the difference between the two is restricted to the first two bytes.

In VMCF terminology, the *source virtual machine* originates the request and the *target virtual machine* processes the request. They are client and server, respectively, in normal parlance.

vmclient is used in the source virtual machine. *vmclisten* and *vmcdata* are used in the target virtual machine. A particular virtual machine can at the same time be target and source, but there can be only one *vmclisten* stage active in a virtual machine at any time.

Supported Functions

Supported functions are *send*, *sendx*, *send/receive*, and *identify*.

Identify

The parameter list is transferred to the target. This completes the transaction. The server cannot reject the message. The *identify* function uses a payload of eight bytes.

Sendx

You can think of *sendx* as *identify* with appended data. The message header and the data are stored as part of reflecting the interrupt. The transaction is then complete. *CMS Pipelines* sets an arbitrary limit of 512 bytes of *sendx* data. As with *identify*, the server cannot reject the message.

Send

At the VMCF level, *send* transmits the parameter list to the message header. The server then inspects the message header and decides whether to receive it or reject it.

When *vmclisten* RECEIVE is specified, a receive operation is performed automatically by *CMS Pipelines*; the output record contains the message header followed by the data received. Except for the length being unrestricted, this form of *send* works like *sendx*.

When RECEIVE is omitted from VMCLISTEN, the output record contains the message header only. Its function code must be modified to VMCPRJCT or VMCPRECV and the record must be passed to *vmcdata* to complete the transaction. Note that the receive function is mandatory with a send function.

Send/receive

The send/receive message, if any, is first received, as you would do for send. The transaction is then completed by passing a VMCPREPL function including reply data to VMCDATA. The receive function is optional with a send/receive function, as it is the reply that completes the transaction.

Parameter lists

CMS Pipelines exposes underlying message headers and parameter lists in the records it produces; and it expects properly formatted parameter lists as input records. While this applies to all three stages, the user is usually concerned with building a parameter list only for *vmclient*, as the input to *vmcdata* is often derived from the output from *vmclisten*.

vmclient: The following fields in the parameter list must be filled in by the producer:

VMCPFLG1	Usually zero.
VMCPFUNC	Function code. Specify VMCPSEND (send, X'0002'), VMCPSENR (send/receive, X'0003'), VMCPSENX (sendx, X'0004'), or VMCPIDEN (identify, X'000A'), as appropriate.
VMCPUSER	Specify the user ID of the target virtual machine unless a user ID is specified as an operand of VMCLIENT.
VMCPLENB	For send/receive, specify the maximum reply size required for send/receive. If specified as zero, the current reply buffer is used; it is at least 4056 bytes.
VMCPUSE	Not inspected or modified by <i>CMS Pipelines</i> . May be used for transaction codes and reasons, as desired by the protocol built on top of the VMCF messages.

VMCPMID, VMCPVADA, VMCPLENA, and VMCPVADA are set by *vmclient* as appropriate; the contents of the input record are ignored.

vmcdata: The function code must be set to:

VMCPRECV (receive, X'0005'),
VMCPREPL (reply, X'0007'), or
VMCPRJCT (reject, X'000B'), as appropriate.

The message ID, user ID, and length fields must remain unchanged from *vmclisten*.

Using VMCF with CMS Pipelines

Figure 254. A Sample VMCF Server Processing

```
/* Process VMCF requests for sharing of the main address space */
Signal on novalue
numeric digits 12
signal on error

/* While data are in the parameter list, we must use send/receive to */
/* be sure that the client is waiting while we perform the adrspace */
/* permit. */

'callpipe (end \ name VMCSERV.REXX:10)',
  '\literal base',
  '|adrspace query',
  '|spec 1.8 c2x 1',
  '|var asit'

say 'vmcserv starting'
'callpipe (end \ name VMCSERV.REXX:6 listerr qualify vmcparm)',
  '\*:',
  '|id: pick m vmcpfunc = m vmcpsenr and m vmcpuse == /Permit /',
  '|o: fanout',
  '|spec x'asit '1 9.8 n',
  '|a:adrspace permit',
  '|hole',
  '\o:',
  '|r:faninany',
  '|vmcdata',
  '|*:',
  '\id:',
  , /* Reject */
  '|spec 1-* 1 m vmcprjct m vmcpfunc', /* Function code */
  '| /reject / m vmcpuse', /* Explanation */
  '|r:'

error:
say 'vmcserv ended. rc='rc
exit RC
```

Example Server Application

This sample causes a server to authorise the client for access to its primary address space. In real life, such a server would likely be rather careful about who it will let peek over its shoulder, but this example shows the raw part only. Figure 254 shows the server code. The server is invoked in a pipeline of this generic form:

```
'PIPE (end \) immcmd stop | stop: faninany | g: gate',
'\vmclisten receive | g: | vmcserv | count lines | stop:'
```

The intent of this arrangement is that you can use an immediate command to terminate the server gracefully by shutting the gate. Any failure in VMCSERV REXX will pass a record to the gate too; it will terminate and thus cause *immcmd* to terminate as well. *vmclisten* can be used in a server that also processes TCP/IP requests. This would typically be done by adding a pipeline to listen on a port and invoke a server:

```

:          'PIPE (end \) immcmd stop | stop: faninany | g: gate',
:          '\vmclisten receive | g: | vmcserv | count lines | stop:',
:          '\tcplisten 1998 | g: | tcpserve ... | count lines | stop:'

```

The workings of the server stage shown in Figure 254 on page 158 are rather simple. If the function code does not indicate send/receive or the user data does not contain “Permit”, the record is passed to the label `id:` where it is turned into a reject parameter list.

Otherwise the record is first turned into the input required to perform the desired `ADRSPACE PERMIT` having the `ASIT` in columns 1 to 8 and the user ID in columns 9 to 16. When the permission has been granted, the record is passed to `vmcdata` to complete the transaction.

This arrangement ensures that the client waits until the permission has been granted. Were an identify function used instead, there would be a race between the two virtual machines; the client may well have tried to create an `ALET` before the server has permitted it.

The client is almost trivial:

Figure 255. Sample Client

```

: pipe literal | spec pad 00 x03 4 /Permit / 33 | vmclient john3 | substr 33-* | cons
: Permit
: Ready; T=0.01/0.01 16:17:20
:
: pipe literal base | adrspace query john3 | alserv add
: Ready; T=0.01/0.01 16:18:35
:
: pipe storage alet 2 2a0 48 | cons
: EXEC          SUTEST FPLA  IMMCMD  EXECDROP
: Ready; T=0.01/0.01 16:19:14
:
: pipe storage 2a0 48 | cons
: PIPE  PIPE  PIPINIT ISREXX  IMMCMD  NUCXDROP
: Ready; T=0.01/0.01 16:19:27

```

First, we send the request to get permission. We then create an `ALET` (see Chapter 18, “Using VM Data Spaces with *CMS Pipelines*” on page 210) to be able to access the other virtual machine’s storage.

Finally, we display the CMS command history information for `JOHN3` and also the one of our own virtual machine, to show that we do access a different address space from our own.

Chapter 15. Event-driven Pipelines in Clients and Servers

So far in this book, we have looked at pipelines performing traditional data processing tasks: reading input from a disk file or a device, processing it, and writing output to a file or a device. *CMS Pipelines* processes data as quickly as it can.

In contrast, this chapter is about event-driven pipelines: pipelines to process commands as they arrive. When using the device drivers described in this chapter, *CMS Pipelines* waits for external events when it has nothing else to do; a stage waiting for an event writes a line to the pipeline when the event occurs. *CMS Pipelines* supplies *starmsg* to capture messages and commands from other virtual machines. *delay* makes things happen at a particular time or after some time.

Though typically running disconnected, a service machine should also be able to process commands when it is connected to a terminal. The person at the console might be an authorised user or the programmer debugging the service machine program. CMS supports immediate commands that can interrupt the running program: you have no doubt issued HI to halt a runaway REXX program. Use *immcmd* to set up an immediate command processor; the argument specifies the name of the command processor to set up. *immcmd* writes a line to the pipeline whenever the user issues the immediate command; the line is a blank character or any string the user types after the immediate command verb.

This chapter may be useful to you even if you have no service machines, you may at times wish to leave your own virtual machine unattended and, for instance, forward notification when a particular file arrives in your reader.

Waking Up Once a Minute

delay copies its input to the output after some time has elapsed or at a particular day and time. *delay* accepts no arguments: it gets the time from input records. The first blank-delimited word on a line specifies the time to wait until *delay* must copy the line to the output; *delay* ignores data after the first word. To issue a CP command once a minute:

Figure 256. Sample delay

```
pipe literal +60 | duplicate * | delay | spec /INDICATE/ 1 | cp | ...
```

The pipeline in Figure 256 generates an infinite number of records with '+60' (but only one at a time).

- *literal* makes one record;
- *duplicate* * copies this record to the output until it gets return code 12 because its output is no longer connected.
- *delay* reads a record, waits for 60 seconds, and writes the record.
- *spec* turns the delay interval into a CP command on each line it reads from *delay*.

Put a *literal* stage (with any argument string) between *delay* and *spec* to generate a command immediately when the pipeline starts.

We have turned the first few stages into a subroutine pipeline:

Figure 257. EVERY REXX

```

/* Write a line after delay.                                     */
signal on novalue
parse arg delay command
if ~abbrev('IMMEDIATE', translate(delay), 3)      /* One right now? */
  Then literal=''                                     /* No...          */
  Else
    Do
      parse arg . delay command
      literal='|literal go!' /* Inject a literal to fire it up */
    End

'callpipe (name EVERY)',
'|literal +delay,          /* Make a relative delay */
'|dup *',                 /* As many as needed     */
'|delay',                 /* Wait                  */
  literal,                /* Fire one immediately, maybe */
'|spec x'c2x(command) '1', /* Turn it into the command */
'|*:'                     /* Pass to output        */

exit RC

```

The *spec* stage shows how to write any string without worry about the stage separator: convert it to hexadecimal.

The time interval begins when the stage after *delay* has processed the output line. Commands are issued less frequently than once a minute if it takes an appreciable time to process the response. You might adjust the delay if the processing always takes the same time. Write a REXX program to take the processing time into consideration; it waits in OUTPUT while *delay* processes a request.

Remember the plus when using an interval. This is also a valid delay: “literal 60|delay”. However, if you issue this command at any time on a Monday, it wakes up at noon on the following Wednesday (because the “60” here means 60 hours after midnight today). This is why EVERY REXX adds the plus.

The first blank-delimited parameter on an input line specifies the time in hours, minutes, and seconds with colons to separate the parts. When there are one or two parts, *delay* assumes zero hours (and minutes) for a relative delay; it assumes zero seconds (and minutes) for time. Specify all three components of the time to be sure. *delay* is not fussy about, for instance, the number of minutes in an hour; +1:67 is the same as +2:7 (or +2:007).

Terminating an Event-driven Pipeline

It remains to sort out one minor detail. When will the pipeline in Figure 256 on page 160 stop? If whatever processes the output from *every* stops, then *every* gets return code 12 writing output. It stops, and so does the rest of the pipeline fragment.

But if nothing will make the output terminate, it will take a while to exhaust the supply of records. *duplicate ** does terminate, at least in principle, after it has written 2147483647 output records. At the rate of one a minute, this will take well over four thousand years and you may not be that patient.

Client/server Pipelines

Issue the immediate command `PIPMOD STOP` from your terminal to terminate *delay* while it waits. You can also force the waiting stages to terminate by passing a record to *pipestop*; it has the same effect. Note, however, that this only stops waiting stages; you cannot terminate a running pipeline this way.

! In a more complicated pipeline with multiple asynchronous stages, you may need a more
! granular way to stop a single waiting stage (as *pipestop* will terminate all waiting stages).
! If you use a *gate* for example to disconnect the primary output stream of *delay* it will
! terminate even while waiting.

Reacting to Immediate Commands

But why not make just `STOP` the way to stop the pipeline? *CMS Pipelines* can process immediate commands: it copies the argument string on the command into the pipeline as you issue the immediate command. *immcmd* writes a line with one blank when it gets no arguments (CMS strips leading blanks from immediate commands).

Figure 258. Delay with Stop

```
'PIPE (end ?)',  
  '?immcmd stop',          /* Output here when user types STOP */  
  '|pipestop',            /* Turn the tap.                */  
  '?literal +60',  
  '|duplicate *',  
  '|delay',  
  '|...'
```

! The more subtle approach to only terminate the waiting *delay* stage would be to use *gate*
! to stop it. The record produced by *immcmd* triggers *gate* to cut the path from its secondary
! input stream to its secondary output stream (identified by the label "g").

Figure 259. Delay with Stop using Gate

```
'PIPE (end ?)',  
  '?immcmd stop',          /* Output here when user types STOP */  
  '|g: gate',              /* Close the gate                */  
  '?literal +60',  
  '|duplicate *',  
  '|delay',  
  '|g:',                   /* Cut the pipe here            */  
  '|...'
```

Let us also make an immediate command to issue CMS commands with full command resolution while the pipeline waits for work. The two commands seem to be useful together; put them in a subroutine pipeline (subroutine pipelines need not be connected to the caller's streams):

Figure 260. ASYNCMS REXX

```

/* Asynchronous CMS commands, pipeline stop */
signal on novalue
'callpipe (end ? name ASYNCMS)',
  '?imcmd stop',          /* Stop commands: */
  '|pipestop',           /* Force shutdown */
  '?imcmd cms',          /* CMS commands: */
  '|subcom cms'          /* Issue to CMS */
exit RC

```

Armed with *asyncms*, write this pipeline instead of the one in Figure 258 on page 162.

Figure 261. Using ASYNCMS with Delay

```

'PIPE (end ?)',
  '|asyncms',
  '?literal +60',
  '|duplicate *',
  '|delay',
  '|...'

```

! For a pipeline that is not supposed to run “forever” it may be necessary to specify the
! INTERNAL option on *imcmd* to make it terminate when all other asynchronous stages have
! terminated.

Processing Messages

Use *starmsg* to get your hands on terminal responses that *cp* and *cms* cannot trap.

You might also consider *starmsg* when CMS’s programmable operator (PROP) does not satisfy your requirements. Maybe you need to preserve state information across calls to the action routine: with PROP you must store state information outside the REXX program, for instance in GLOBALV. In contrast, your action routines run concurrently with *CMS Pipelines*.

Select the type(s) of message to process with the CP command SET. The easiest service machine to set up processes commands from users on the same system, sent with the CP command MSG. More sophisticated servers can service requests forwarded as RSCS messages. Here is an example of the first kind:

Figure 262.

```

'CP SET MSG IUCV'
'PIPE starmsg | spec 9-* 1 | validate | spec 9-* 1 | subcom cms'

```

Lines from *starmsg* have eight bytes with the type of message followed by eight bytes with the origin user ID followed by the message data, if any. In this case the message type prefix is the same on all lines: discard it. *validate* (which is shown in Figure 265 on page 165) ensures that only those we trust get service. Use a decoding network to process requests from users in particular ways. *starmsg* sets up the immediate command HMSG to make it stop. Issue HMSG (or PIPMOD STOP) from the terminal to terminate *starmsg*.

Client/server Pipelines

Figure 263. Message Classes

Class	Enabled By SET	Message Source
1	MSG IUCV	Messages sent with the CP command MESSAGE (MSG) or MSGNOH.
2	WNG IUCV	Warnings sent with the privileged CP command WARNING (WNG).
3	CPCONIO IUCV	Synchronous command responses; echo of terminal input; asynchronous responses not presented by other means.
4	SMSG IUCV	Messages sent with the CP command SMSG.
5	VMCONIO IUCV	Virtual machine generated output, for instance from the REXX Say instruction or the <i>console</i> device driver.
6	EMSG IUCV	CP error messages.
7	IMSG IUCV	CP informational messages.
8		Terminal output routed through Single Console Image Facility from a machine for which this machine is the secondary user. You cannot disable this message class.

Warning: Setting CPCONIO IUCV means that **all** console output generated by CP is presented to you. This includes the echo of commands you type on the terminal; they are indistinguishable from CP responses. You also receive messages and warnings when the corresponding setting is ON.

Figure 264. Sample STARMMSG

```

pipe literal set cponio iucv | cp set msg on | console
▶Ready;

pipe starmsg | console
▶00000003JOHN    "CP MSG * HI, THERE!
▶00000003JOHN    22:08:42""MSG FROM JOHN    : HI, THERE!""
▶00000003JOHN    "CP SET MSG IUCV
▶00000003JOHN    "CP MSG * HI, THERE!
▶00000001JOHN    HI, THERE!
▶00000003JOHN    hmsg
▶Ready;

```

The double quote characters represent line end characters (X'15'). We entered four commands while *starmsg* was intercepting console output:

```

#cp msg * Hi, there!
#cp set msg iucv
#cp msg * Hi, there!
hmsg

```


CP translates the echo of CP commands to upper case, but not the echo of CMS commands. Try to match the four commands to the responses in Figure 264. Note the change in the message class prefix after MSG was set to IUCV; after this, the message is no longer treated as CP-generated console output; also note that there is no time stamp.

Validating a User ID

If the file VALID USERS contain the user IDs of the users who are allowed to access a server, you can use *lookup* to filter those that are not authorised:

Figure 265. Validating Clients

```

/* Validate user ID in cols 1.8                                     */
'callpipe (end ? name validate)',                                */
'|*:',                                                         /* From STARMSG without class */
'| |1: lookup 1.8 details',                                     /* Validate them               */
'|*:',                                                         /* These are OK                */
'?< valid users',                                           /* Friends                      */
'|split',                                                    /* One per line                 */
'|pad 8',                                                    /* Make full length            */
'|1:',                                                       /* To lookup                    */
'|change //Access violation: /',                             /* A comment added.           */
'|console'                                                  /* Display it.                 */

```

Chapter 16. *spec* Tutorial

This chapter is a tutorial on *spec*, the Swiss Army Knife of *CMS Pipelines*. You will see how to use *spec* for many diverse applications.

spec has evolved from a simple filter to a complex programming language, but the language can be subset: You can choose a subset you wish to learn; you do not have to learn about the other features just to avoid them.

As you progress through this tutorial, you will realise that some of the statements made in the early sections might be in need of the odd qualifying footnote. However, if you choose a subset that does not include the finer points, you do not need to know these finer points and a sprinkling of footnotes becomes a nuisance rather than a help.

You will find a concise reference for *spec* in Chapter 24, “*spec* Reference” on page 719. Refer to that right now if you prefer to read a complete authoritative reference rather than a tutorial.

The examples in this chapter are formatted with the *spec* stage across the entire column and the input records below to the left and output records below to the right. To make reading easier, each specification item is on a separate line. For reasons of typography, it is not possible to put meaningful headings into this layout; you will have to remember that the left hand side contains the input records and that the result is shown on the right. The good news is that the examples are run when the book is formatted for printing. What you see is indeed what it does, even when the examples contain mistakes. This printing applies to:

Figure 266. Pipeline Level Used for Examples

```
pipe query level
►CMS Pipelines, 5741-A07 level 110C0011
►Ready;
```

Basic Mechanics

spec reads an input record; it then interprets its argument string and produces an output record when it reaches the end of the argument string. It then repeats this *cycle* with each new input record until it reaches end-of-file.

The argument list to *spec* is called a specification list, because it is interpreted as a list of *specification items*. Some specification items are keywords that control how *spec* operates; others define the contents of fields in the output record.

spec processes the specification list from left to right, but the output record need not be built from left to right; a specification item can modify a part of the output record that has already been filled by a previous specification item.

Basic Field Handling

A field in the output record can contain data from a field of the current input record, constant literal data, or data generated within *spec*. First we look at specification lists that build records from input fields and literal data.

Input Ranges

The basic specification item copies part of the input record to the output record. It is specified as an *inputRange* followed by a *number*.

To copy a record unchanged from the input to the output:

Figure 267. Copy Record Without Change	
specs 1-* 1	
First record Second record	1 1 ds1 First record Second record

The specification list in Figure 267 contains a single specification item. This item contains an input range (1-*) and an output column number (1).

An asterisk in an input range is interpreted as the beginning of the record when it is first and as the end of the record when it is after the hyphen; thus, both **** and *1-** specify the entire record.

To select a subset of the input record and indent it in the output record:

Figure 268. First Word Only	
specs word 1 5	
First record Second record	First Second

Figure 268 shows that an *inputRange* can select things other than just columns. WORDS (which can be abbreviated to *w*) specifies that the range refers to blank-delimited words. In this example the first word of each input record is inserted in the output record beginning in column 5. The first four columns are filled with blanks.

When you specify a word range, *spec* interprets that as the range of columns from the beginning of the first word to the end of the last one. It does not squish out multiple blanks within such a range:

<i>Figure 269. Some Words</i>	
<pre>specs word 2-4 5</pre>	
<pre>Here is the first record Here is the second record</pre>	<pre>is the first is the second</pre>

In this example you can remove the excess blanks easily; just do the three words one at a time:

<i>Figure 270. Some Squished Words</i>	
<pre>specs word 2 1 word 3 nextword word 4 nextword</pre>	
<pre>Here is the first record Here is the second record</pre>	<pre>is the first is the second</pre>

NEXTWORD (which can be abbreviated to NEXTW and even to NW) specifies that the field is appended to the contents of the output record after a blank is added as a separator. The blank is omitted when the output record is empty.

Note that you must specify WORD for each specification item that refers to a word range; this will allow you to refer to words in some specification items and to columns in others.

spec also supports tab-delimited fields. Just as words are separated by blanks, fields are separated by horizontal tabulate characters (X'05'). But whereas words can be separated by more than one consecutive blank, two adjacent tabulate characters have a null field between them (that is, a field of length zero).

You can specify a different tabulate character with the FIELDSEPARATOR keyword (or its synonym FS). To move the contents of fields that are delimited by equal signs to specific columns:

<i>Figure 271. Selecting Fields</i>	
<pre>specs fieldseparator = field 1 1 field 2 6 field 3 11 field 4 16</pre>	
<pre>a=b =a ==a=b</pre>	<pre>a b a a b</pre>

Notice that the first two records contain only two fields; the third record contains four fields; the second and third records contain null fields.

An *inputRange* can contain a negative number; this specifies that the count is from the end of the record rather than from the beginning:

<i>Figure 272. Penultimate Word</i>	
specs word -2 1	
First record is long Second record	is Second

The general form of a range consists of two numbers separated by a semicolon. Thus, there is a third idiom to refer to the entire record: 1;-1. When both numbers are positive, there is no difference between using semicolon and using a hyphen to delimit the numbers.

When the two numbers have the same sign, the first number must be less than or equal to the second one; it is an error to specify an ending column that is before the beginning one. (Recall that -2 is less than -1.) When the numbers have different signs, a null input field is used when the beginning position is after the end position:

<i>Figure 273. Word Range</i>	
specs word 2;-2 1	
First record is long Second record Third one even more words	record is one even more

The second output record is a null record (it contains no data), because the field to be written started at the beginning of the second word (the first “r” of “record”) and extended to the end of the second last word (the first “d” of “Second”). Since the input field ends before it begins, the output field is null. (Null records are not written to CMS files, because CMS does not support null records in files that have variable record format. But *spec* produces a null record all the same.)

When the first number in a range is positive, you can specify a count rather than the last number. The count is specified after a period and it must be positive:

<i>Figure 274. Some Words, Revisited</i>	
specs word 2.3 5	
Here is the first record Here is the second record	is the first is the second

Substrings: You can refer to a substring of a range, to a substring of a substring of a range, and so on:

<i>Figure 275. Using Substring</i>	
specs substring 2;-2 of word 3 1	
This is the first record and the second record	h econ

CMS Pipelines processes the input range from right to left. It starts with the complete record. It then processes word 3; this string becomes the input record for the substring expression. You can mix fields, words, and column ranges within a substring expression; you can even have different field separators and word separators for different parts of the expression.

To find the variable or stem being assigned in a REXX assignment instruction:

<i>Figure 276. Using Substring</i>	
specs substring fieldsep . field 1 of substr word 1 of fs = f 1 1	
a = 17 x.18= 23	a x

: **Structured data:** Rather than referring to the absolute column, word, or tab-delimited
: field, you can declare structures that contain members, as described in Chapter 6, “Proc-
: essing Structured Data” on page 91 and in the description of *structure*.

: Such structure definitions can be created manually or possibly by a utility from an already
: existing machine readable record layout, or even dynamically.

: In this chapter we shall use the structure in Figure 277 to show examples of the use of
: structured data.

```

: pipe < samp record | console
: ▶:str mem 1.4
: ▶ char c len 4 bin d len 4
: ▶ float f len 8 pack p(2) len 4
: ▶Ready;

: pipe < samp record | struct add thread
: ▶Ready;

: pipe struct list str | console
: ▶:str <length 24>
: ▶ mem 1.04
: ▶ char C 5.04
: ▶ bin D 9.04
: ▶ float F 13.08
: ▶ pack P(2) 21.04
: ▶Ready;

```

Figure 277. A Sample Structure

: Read the literal as: Structure `str` contains an member named `mem`, which is four bytes and
 : has no type associated, beginning in the first column. The next four columns contain
 : member `char`, which is of character type as indicated by the single character. The next
 : four columns contain member `bin`, which is a binary number in two's complement nota-
 : tion as indicated by the type `D`. The next eight columns contain member `float`, which is
 : a System/360 hexadecimal floating point number. The final member of the structure is
 : `pack`, which is a packed decimal number. The length four accommodates seven digits and
 : a sign. The scale is two (there are two decimals in the number); this is the data type
 : known as computational-3 to COBOL programmers. Structure and member names are case
 : sensitive.

: As binary data are cumbersome to construct and also not to obscure examples by creating
 : such numbers, we have prepared a two record file, which is dumped in Figure 278.

: The important point is that, except for data typing and scaling of packed decimal data, a
 : member of a structure is simply a symbolic way to specify a particular substring of the
 : input record; thus most of our examples will show column numbers as that keeps the
 : example compact, but you should use structures and members for production.

: The utility of structures comes, of course, when the record layout changes; you no longer
 : need to track down the various EXECs that are affected by a change.

```

pipe < struct data | fmtcmp | console
▶0Record 1
▶ 00000000 998583F1 8699A2A3 00000011 413B3333 *rec1frst *
▶ 00000010 33333333 * *
▶0Record 2
▶ 00000000 998583F2 A2839584 00000015 42118000 *rec2scnd â Ø *
▶ 00000010 00000000 * *
▶Ready;
  
```

Figure 278. Sample File Containing Structured Data

<i>Figure 279. Example of Using Structured Data</i>	
specs member str.char 1	
(Contents of STRUCT DATA.)	frst scnd

This reads the sample file, selects member `char`, and prints it.

Literals

To add a literal string to each record:

<i>Figure 280. First Word and a Literal</i>	
<pre>specs w1 1 /banana/ nextword</pre>	
First record Second record	First banana Second banana

In Figure 280, the first word is inserted in column 1 and the literal string is appended after a blank. WORD can be abbreviated down to w and you can elide the blank between the keyword and the word number.

A literal can also be expressed as a string of hexadecimal digits or a string of binary digits:

<i>Figure 281. Hexadecimal Literal</i>	
<pre>specs w1 1 x5c next w2 next</pre>	
First record Second record	First*record Second*record

NEXT specifies that the field should be abutted to the contents of the output record so far. (X'5C' is the hexadecimal representation of the asterisk.)

Manifest Constants

A manifest constant is also literal data, but it is four bytes binary. Typically, a manifest constant is used to insert a particular value in a control block or parameter list being built.

```
pipe struct list vmcparm | ...
... pick from w1 == /vmcpfunc/ to after substr 1.8 of w1 == /vmcpsend/ |
... console
▶ vmcpfunc D 3.02
▶ vmcpauth=0
▶ vmcpuaut=1
▶ vmcpsend=2
▶Ready;
pipe spec qualify vmcparm eof m vmcpsend m vmcpfunc | ...
... spec 1-* c2x 1 | console
▶40400002
▶Ready;
```

The manifest constant is a binary constant of four bytes. It is by default entered into the output field aligned to the right. (Using EOF is a handy way to force spec to generate a null input record internally.) In this case, the output field is two bytes starting in column 3; hence the two leading blanks.

The Record Number

To insert the record number in the first ten columns:

<i>Figure 282. Number, First Word, and Literal</i>	
<pre>specs number 1 word -2 nw /banana/ nw</pre>	
First record	1 First banana
Second record	2 Second banana

The keyword NUMBER refers to a field maintained by *spec*. The field is ten characters wide; the number is aligned to the right with leading zeros suppressed.

: Originally NUMBER was just that, the record number; and there was just one counter for all
 : NUMBER items. This changed, however, when increment and starting number were added
 : around 1989.

: Nowadays, there is a counter for each NUMBER item. This counter is incremented each
 : time the item is issued. In simple specification lists, this translates to the same as the
 : original definition, but add conditionals and the number may no longer be the same as the
 : record number.

Output Placement

So far, the output field has contained precisely the characters in the input field.

You can specify the size of the output field to make it shorter or longer than the input field. The input field will be padded with blanks or truncated, as required to fill the width you have specified:

<i>Figure 283. First Word Chopped</i>	
<pre>specs w1 1-3 /*/ next</pre>	
First record	Fir*
Second record	Sec*

: And you can even specify the output as a member of a structure:

<i>Figure 284. Specifying Output Position as a Member</i>	
<pre>specs w1 member str.char</pre>	
First record	Firs
Second record	Seco

With a *placement option*, you can control how the field is inserted into the output record. You can align the field to the left or to the right; or you can centre it. When you use a placement option, the input field is stripped of blanks before it is placed. To put the sequence number into columns one through five aligned on the left:

<i>Figure 285. Number and Literal</i>	
<pre> specs number 1.5 left /*/ next </pre>	
First record	1 *
Second record	2 *

Figure 285 also highlights the fact that you need not copy any input fields to the output record; you still get as many output records as there are input records.

An input field that does not exist in a particular input record is considered to be null; that is, it contains no characters. When a null input field is referenced in a specification item that does not specify an explicit length, the specification item is ignored. In particular, the output record is not padded to the position of the output field:

<i>Figure 286. Null Input Field Ignored</i>	
<pre> specs word 1 1 word 3 10 /*/ next </pre>	
First record	First*
Second record longer	Second longer*

Padding

When an output field is placed beyond the current end of the output record, the gap is filled with the *pad character*, which is also used when output fields are placed with a particular length.

You can use PAD to change the pad character to use in subsequent specification items:

<i>Figure 287. Padding with Asterisks</i>	
<pre> specs word 1 5 pad * word 2 15 </pre>	
First record	First*****record
Second record	Second****record

The first word is inserted after four blanks, because the blank is the default pad character and it has not yet been overridden; the second word is inserted after five asterisks in the first record and after four asterisks in the second record.

You can change the pad character as often as you like.

You can resort to a subterfuge to put the record number in the first five columns and insert leading zeros:

Figure 288. Number and First Word	
<pre> specs pad 0 number 1.5 right w1 nextword </pre>	
First record	00001 First
Second record	00002 Second

Rather than supply an operand to specify no leading zero suppress (there is no such operand), you can use PAD to specify the pad character to be used when the stripped number is inserted into the output record; thus, the net effect is the one desired. Note that NEXTWORD inserts a blank irrespective of the setting for the pad character.

Conversion

: This section discusses explicit conversion. When you define structures with typed
 : members, their contents are converted automatically to the desired form; you may not
 : specify explicit conversion too.

Conversion can be used to make binary data visible as well as to turn printable data into the *internal representation*, which is the form numbers have inside the computer.

Conversion to printable form is often used with *sql* SELECT. Here, however, is a *spec* stage that formats the first eight characters of a line into a form often used by programmers:

Figure 289. Make Data Printable	
<pre> specs 1.4 c2x 1 5.4 c2x nextword /*/ 19 1.8 next.8 /*/ next </pre>	
First record	C68999A2 A3409985 *First re*
Second record	E2858396 95844099 *Second r*
Short	E2889699 A3 *Short *

The conversion used in the example in Figure 289 unpacks a byte of data into two bytes in hexadecimal notation. The eight bytes of input data are split into two fields, which are printed with a blank between them. You can see that a short input field is not padded before conversion.

The input data are placed in the output record a second time, this time without conversion. An asterisk is inserted in column 19 followed by eight bytes of input data and a closing asterisk. It would be normal to translate any unprintable characters in the original record to blanks in a subsequent *xlate* stage.

Combining Input Records into One Output Record

You can easily process a pair of input records and produce a single output record for the two input records. Use `READ` or `READSTOP` to read another input record in the middle of the specification list. Of course, once you have read a new record, the previous one is gone and you can no longer refer to it.

To merge the words of each pair of input records:

<i>Figure 290. Merging Words from Two Input Records</i>	
<pre> specs word 1 1.4 word 2 11.4 word 3 21.4 read /*****/ 26 word 1 6.4 word 2 16.4 word 3 26 </pre>	
<pre> First record here Second line follows Odd record </pre>	<pre> Firs Seco reco line here follows Odd reco ***** </pre>

`READ` assumes a null record when it gets end-of-file. `READSTOP`, in contrast, terminates the specification list:

<i>Figure 291. Merging Words from Two Input Records</i>	
<pre> specs word 1 1.4 word 2 11.4 word 3 21.4 readstop /*****/ 26 word 1 6.4 word 2 16.4 word 3 26 </pre>	
<pre> First record here Second line follows Odd record </pre>	<pre> Firs Seco reco line here follows Odd reco </pre>

Multiple Input Streams

You can combine data from several sources, because *spec* can process any number of input streams concurrently. This can be used to generate a listing of two files side by side:

Figure 292. Side by Side

```
'PIPE (end ? name PIPUSPET.SCRIPT:342)',
  '? < first file',
  '|s: spec',
      '1-*    1',
      'select  1',
      '1-*    40',
  '| > output file a',
  '? < second file',
  '|s:'
```

The multistream support in *spec* follows the same pattern as the multistream support in a REXX pipeline filter. That is, you first select the stream you wish to read from. Then you can use the normal read operation to read from the stream.

spec is different in one respect, however. It *synchronises* its input streams before it starts processing a set of input records; and it consumes the set when it comes to the end of the specification list.

The synchronisation operation ensures that all records are available; and the synchronisation operation is easy to understand. But if two input streams originate in a common stage, for example *chop*, the pipeline is likely to stall unless you take precautions. Refer to “Ensure the Pipeline Does not Stall” on page 88.

Generating Several Output Records from One Input Record

Just as you can combine a pair of input records (or more) with READ, you can generate more than one output record while processing a single input record. Use WRITE to write a record containing the data generated so far:

Figure 293. Writing Many Output Records

<pre>specs word 1 1 number 15.5 right /#1/ 12 write word 2 1 /#2/ 12 number 15.5 right</pre>			
First record	First	#1	1
Second line	record	#2	1
Short	Second	#1	2
	line	#2	2
	Short	#1	3
		#2	3

Multiple Output Streams

spec can write to all connected output streams. This is accomplished by `OUTSTREAM` in the same way an input stream was selected with `SELECT`. You would typically use `WRITE` to write the record, when producing output on two streams:

Figure 294. Writing Two Output Streams

```
'PIPE (end ? name PIPUSPET.SCRIPT:448)',
  '? < input file',
  '|s: spec',
      'word 1 1',
      'write',
      'word 2 1',
      'outstream 1',
  '| > first words a',
  '?s:',
  '| > second words a'
```

The pipeline in Figure 294 produces two output files, both containing the same number of records as in the file `INPUT FILE`. The file `FIRST WORDS` contains the first word of each input line; `SECOND WORDS` contains the second word of each input line.

Unlike `SELECT`, `OUTSTREAM` takes effect when the record is written, not when data are placed in the output record. You can build only one output record at a time.

Expressions

spec performs decimal arithmetic with thirty-one digits precision. You can save the result of a calculation in a *counter*, where it can be stored for use in a subsequent record.

You can format the contents of a counter for printing under control of a *picture*, which is a pictorial representation of the formatting you require.

You can suppress the automatic writing of an output record so that a record is written only at end-of-file.

Counter Expressions

A counter is identified by a number that is zero or positive. The syntax to specify a counter consists of a number sign (`#`) followed by the number of the counter; for example, `#17`.

The code point for the number sign is `X'7B'`, which displays the number sign on an English terminal; however, not all terminals display this code point the same way. (The character is also called a hash or a pound sign, but this must not be confused with the currency symbol for pound sterling.) If there is a number sign on your keyboard, you can go ahead and use it. If your keyboard does not have a number sign, some other character must be used. On a French keyboard, the pound sterling symbol would probably work. It is easy to find out with this pipeline:

Figure 295. Finding the Character Displayed for 7B

```
pipe literal 7b | spec w1 x2c 1 | console
▶#
▶Ready;
```

The number of the counter is specified after the number sign. Counters are numbered from zero and upwards. *spec* can store values in as many counters as you need; there is no arbitrary limit to the number of counters.

The values stored in counters may be numbers or strings, so the name counter is slightly misleading; maybe *register* would be more appropriate, but the old name sticks.

The Arithmetic/Logic Unit (ALU) implemented by *spec* can reference data in input records in three ways:

- Using MEMBER, as we have seen already.
- Indirectly through a *field identifier* that specifies the field that contains the number to use. Syntactically, a field identifier is a single letter followed by a colon; it is placed in front of the *inputRange*. Case is respected in field identifiers; thus, there are fifty-two possible field identifiers.
- Through the record function, as we shall see when we get to expressions.

The following example of using the *spec* ALU sums the values of the first word of each input record and prints a running total in each line:

Figure 296. Summing

```
specs
  1-*      1
  a: word 1 .
  set #0:=#0+a
  print #0 10
```

1 one	1 one	1
2 two	2 two	3
3 three	3 three	6

In the example in Figure 296, the first specification item (1-* 1) simply copies the input record to the output record. The second item (a: word 1 .) associates the field identifier a with the first word in each record, but it does not place that word in the output record, because the placement is specified as a period, which means “ignore”.

The third specification item (set #0:=#0+a) can be read as *set counter zero to the sum of its current contents and the contents of field a*. Thus, this item accumulates the running total in counter 0. := is the *assignment operator*. Note the colon, which distinguishes this operator from the = relational operator, which tests for numeric equality.

The fourth specification item (print #0 10) “prints” the contents of counter 0. That is, it places the contents of the counter into eleven characters starting in column 10 of the output record. By default, PRINT formats the value with leading zeros “suppressed” (converted to blanks); you can control the formatting with a picture, as we shall see later.

You can also refer directly to members of structures. Members that have a numeric type are converted automatically when assigned to a counter. This applies whether you refer-

:
: ence the member directly or you use a field identifier to reference indirectly a range that is
: defined by a member.

<i>Figure 297. Summing with Structured Data</i>		
<pre> specs print str.bin 1 set #0:=#0+str.bin print #0 n </pre>		
(Contents of STRUCT DATA.)	17	17
	21	38

:
: Rather than supplying the fully qualified member name each time you refer to a member
: of a particular structure, you can declare a *qualifier* for the current stream:

<i>Figure 298. Using a Qualifier</i>		
<pre> specs qualify input str 1 print bin 1 set #0:=#0+bin print #0 n </pre>		
(Contents of STRUCT DATA.)	17	17
	21	38

:
: You can also specify in which column the structure should start; here we have specified
: the default explicitly (this is a good habit to get into).

:
: You can specify a separate qualifier for each input and each output stream and you can
: also specify that a qualifier should apply to all input streams, all output streams, or all
: streams.

The example in Figure 296 on page 179 was cast for the reader who is familiar with REXX. It can be written more compactly by using operators borrowed from C. For example, the SET #item is redundant; the counter can be updated as it is printed:

<i>Figure 299. Summing while Printing</i>		
<pre> specs 1-* 1 a: word 1 . print #0+=a 10 </pre>		
1 one	1 one	1
2 two	2 two	3
3 three	3 three	6

This example uses the increment operator (+) to add the contents of the identified field to the contents of the counter. This is the preferred way to increment a counter before it is “printed”.

String Processing

The example in Figure 299 on page 180 can be made even more compact using some of the string functions:

<i>Figure 300. Summing while Printing</i>		
specs		
1-*	1	
print #0+=word(record(), 1) 10		
1 one	1 one	1
2 two	2 two	3
3 three	3 three	6

record() returns the entire input record; and word selects the first blank-delimited word; finally, the assignment with add forces conversion of the string "1" to a number, which is added to the contents of the counter.

You can store input string data in a counter and you can concatenate strings and apply most of the REXX functions to strings.

For example, to reverse the third word of the input record:

<i>Figure 301. Reversing the Third Word</i>	
specs	
a: 1-*	1
b: word 3	.
print reverse(b) (max(1, wordindex(a, 3)))	
"The time has come," the Walrus said, "To talk of many things: Of shoes--and ships--and sealing-wax-- Of cabbages--and kings-- And why the sea is boiling hot-- And whether pigs have wings."	"The time sah come," the Walrus ,dias "To talk fo many things: Of shoes--and dna--spihs sealing-wax-- Of cabbages--and --sgnik And why eht sea is boiling hot-- And whether sgip have wings."

The output position in this example is a *computed output position*. The wordindex function very conveniently provides the position of the third word, at least when there is one. Finally, max guards against the case where there are two or fewer words in the record as the word index is zero in this case.

You should also note that you can print a string, but you must not supply a picture; doing so would force conversion to a number.

Here is an example of concatenating strings (the OR bars are doubled to escape them):

Figure 302. Catenating Strings	
<pre> specs a: w1 . b: w2 . set #0 =a set #1 =b print #0 " and " #1 1 </pre>	
<pre> 1 one 2 two 3 three </pre>	<pre> 1 and one 12 and onetwo 123 and onetwothree </pre>

You can make an expression more readable by putting it in parentheses, because you can then sprinkle blanks into it. The previous print item could be written as:

```
print ( #0 |||" and " |||#1 ) 1
```

But it could not have been written like this, because *spec* has not implemented the blank operator that REXX uses to concatenate with a blank:

```
print ( #0 "and" #1 ) 1
```

Dealing with Errors in Expressions

The *spec* expression parser implements a rather rich language using what is known as *bottom up parsing*. The good thing about such a parser is that there are compiler generators that can construct the parsing tables, which makes the whole thing manageable. The downside is that errors are reported from the parser’s point of view, which is not always easy to understand. As an example, let us try the erroneous expression above:

```

pipe spec print (#0 "and" #1) 1
►Parse error in state 80, unexpected T_QSTRING at offset 4: ""and" #1) 1"
►... Issued from stage 1 of pipeline 1
►... Running "spec print (#0 "and" #1) 1"
►Expecting S_RP S_SEMI S_EOD
►... Scan at position 15; previous data "print (#0 "and"
►Ready(01434);
    
```

Figure 303. Parser Errors

The parser is trying to tell you that it does not like two abutted terms. The state number (80) has meaning only to the programmer who built the parser (because he can refer to a listing that defines the state, which is assigned by the compiler generator). The parser then informs you that it is expecting to see a right parenthesis, a semicolon, or the end of the expression (at least, so the programmer would tell you—you might not be quite that clairvoyant). You might also wonder why it does not tell you that you should use the concatenate operator when it just accepted such a construct in the previous example, but such are the ways of LALR(1) parsers (for that is what it is). The good news is that the programmer has added many rules for erroneous syntax to the grammar to issue meaningful error messages, but eventually you will arrive at “crunch point” where the LALR(1) parser rears its head.

Special Processing at End-of-file

To print the total at end-of-file:

<i>Figure 304. Summing</i>	
<pre> specs 1-* 1 a: word 1 . set #0+=a eof /Total:/ 1 print #0 next </pre>	
<pre> 1 one 2 two 3 three </pre>	<pre> 1 one 2 two 3 three Total: 6 </pre>

As each record is processed, counter 0 is incremented by the contents of field a, as before. As long as *spec* has an input record, it stops processing the specification list when it encounters the EOF specification item. When *spec* reaches end-of-file, it processes the specification items that follow EOF. These two items format the contents of counter 0 to print a summary record.

Note that there are more output records than input records, even though there is no WRITE item. The reason is that *spec* performs an additional final cycle when it reaches end-of-file. It takes this *runout cycle* to process the specification items after EOF.

You can suppress output for all detail records and print only the summary record at end-of-file:

<i>Figure 305. Summing Quietly</i>	
<pre> specs printonly eof a: word 1 . set #0+=a eof /Total:/ 1 print #0 next </pre>	
<pre> 1 one 2 two 3 three </pre>	<pre> Total: 6 </pre>

PRINTONLY EOF specifies that no output records are to be written until the runout cycle.

You can include the count of records:

<i>Figure 306. Summing and Counting</i>	
<pre> specs printonly eof a: word 1 . set #0+=a set #1+=1 eof /Total:/ 1 print #0 strip nextword /in/ nextword print #1 strip nextword /records./ nextword </pre>	
<pre> 1 one 2 two 3 three </pre>	<pre> Total: 6 in 3 records. </pre>

During each cycle, the contents of field a are added to counter 0 and the constant 1 is added to counter 1. Thus, at end-of-file, counter 0 contains the total, as before, and counter 1 contains the record count. The specification items following EOF display the contents of these counters.

You can use STRIP to strip all types of input fields of leading and trailing blanks. In Figure 306, it is used to strip the leading blanks from the counter being printed.

You can even combine the two SET specification items into one by using the *discard operator*:

<i>Figure 307. Summing and Counting</i>	
<pre> specs printonly eof a: word 1 . set (#0+=a; #1+=1) eof /Total:/ 1 print #0 strip nextword /in/ nextword print #1 strip nextword /records./ nextword </pre>	
<pre> 1 one 2 two 3 three </pre>	<pre> Total: 6 in 3 records. </pre>

The SET specification item contains two expressions that are separated by the semicolon operator. In the example in Figure 307, it works like the REXX clause delimiter, because the result of the expression is discarded.

The expression is enclosed in parentheses to make it more readable. This allows the use of blanks to separate the terms of the expression; without the parentheses it must be written as #0+=a;#1+=1.

Pictures

Try twiddling the input data to explore the numeric range supported by *spec*:

<i>Figure 308. Big Sums</i>		
specs		
1-*		1
a: word 1		.
print #0+=a		16
1	1	1
-3	-3	-2
20.6	20.6	18
3e2	3e2	318
4.7e5	4.7e5	470318
4.7	4.7	470323

As you can see, *spec* does not complain about decimal fractions. A counter can hold floating point numbers with up to thirty-one decimal digits of precision. The exponent range is in the thousand millions, which should be *quite* sufficient for most needs.

If you study the numbers and the results in Figure 308 carefully, you will see that the computation has been performed without loss of precision, but printing has truncated the number to an integer. You can specify a *picture* to control the way the contents of a counter are formatted. A picture is a string of characters that specifies the desired format; this string contains one character for each column of the formatted field. The picture is specified after the keyword PICTURE. Case is ignored in pictures.

999 is a simple picture, which specifies that the number is to be formatted as three digits, with no sign and no decimal point and no suppression of leading zeros. If, for example, counter 4 contains the value 16 and the specification item is `print #4 picture 999 1`, the output field will be 016. To get suppression of leading zeros, use `z`, rather than `9`, in your picture. In this case, if the picture is changed to `zz9`, the output becomes 16. To allow for negative numbers, use one or more minus signs in the picture. For example, if the counter contains the value -16 and the picture is `---9`, the output will be -16. The minus sign is said to *drift*; it is replaced by blanks until just before the first nonzero digit in the output.

If you omit the PICTURE keyword, *spec* uses a default picture that has a drifting minus sign with ten digits and no decimal fraction. Hence the truncation in Figure 308.

Use a decimal point to print fractional digits:

<i>Figure 309. Summing and Averaging</i>	
<pre> specs printonly eof a: word 1 . set (#0+=a; #1+=1) eof print #1 1 /observations./ nextword write print #0 1 /total./ nextword write print #0/#1+.005 picture -----9.99 1 /average./ nextword </pre>	
<pre> 1 one 2 two 3 three 4 four </pre>	<pre> 4 observations. 10 total. 2.50 average. </pre>

This example is a variation on Figure 306 on page 184.

The second last specification item computes the average by dividing the total by the count of observations. The result is increased by five thousandths to ensure correct rounding when the number is truncated for formatting with the picture.

This particular picture specifies eight hyphens, which represent a *drifting sign*; a nine, which represents a digits position; a period, which represents the units position as well as the character to insert for the decimal point; and two more nines to represent the first two digits of the decimal fraction. For negative results, a hyphen is inserted into the last position that contains a blank.

Conceptually, the picture is processed by first converting the result of the expression to a number that has eight digits before the decimal point and two digits after the decimal point. That is, the number has two digits fewer than the number of characters in the picture, because the drifting sign and the period each require a position. The digits in this string are then inserted into the output record under control of the picture. For the hyphens making up the drifting sign, leading zeros are suppressed and replaced by blanks. The character 9 indicates that the digit is to be inserted unconditionally. Thus a number numerically less than one will have a zero digit just in front of the decimal period.

Use STRIP to format counters into fields of variable sized:

<i>Figure 310. Summing and Averaging</i>	
<pre> specs printonly eof a: word 1 . set (#0+=a; #1+=1) eof print #1 strip 1 /obs;/ nextword print #0 strip nextword /tot;/ nextword print #0/#1+.005 picture zzzzzz9.99 strip nextword /avg./ nextword </pre>	
<pre> 1 one 2 two 3 three 4 four </pre>	<pre> 4 obs; 10 tot; 2.50 avg. </pre>

There is no picture character to suppress trailing zeros.

Using counters and pictures, the record numbering shown in Figure 288 on page 175 can be accomplished in a much simpler way:

<i>Figure 311. Number and First Word with Pictures</i>	
<pre> specs print #0+=1 picture 99999 1 </pre>	
<pre> First record Second record </pre>	<pre> 00001 00002 </pre>

The picture in Figure 311 inserts five digits unconditionally.

Try running some numbers through a picture:

Figure 312. Big Sums with Fractions	
<pre> specs a: word 1 1 print a picture *,**,**9.99s 8 print #0+=a picture ++++++9.99 nw </pre>	
1	1 *****1.00+ +1.00
-3	-3 *****3.00- 2.00
20.6	20.6 *****20.60+ +18.60
3e2	3e2 *****300.00+ +318.60
4.7e5	4.7e5 **470,000.00+ +470318.60
4.7	4.7 *****4.70+ +470323.30
47E10	Counter contains more digits than ... Issued from stage 2 of pipelin ... Running "specs a: word 1 1 pri Processing item number 2: print a

The first word of the output record contains the input field.

The second word shows the number printed with *cheque protection* where asterisks rather than blanks are used to suppress leading zeros. The commas in the picture are displayed as commas if the number has started. The decimal fraction is displayed with two decimal places. The s character specifies that the sign should be inserted after the number. Zero is considered positive.

The third word in the output shows the running total printed with a drifting plus sign. This example is shown here to warn you that a drifting plus results in a negative value being formatted with no sign. Use a drifting s to prefix a number with either a plus or a minus.

The last number is too large to print using the picture specified; but the number is well within the range you can store in a counter.

You can use scientific notation for expressions that have a very large range of potential values:

<i>Figure 313. Scientific Picture</i>	
spec a: word 1 . print a picture s9.99999es999 1	
0	+0.00000e+000
1	+1.00000e+000
10	+1.00000e+001
-00000000000001	-1.00000e+000
-0.0000000000001	-1.00000e-012
.0034	+3.40000e-003
17e-4	+1.70000e-003
10000000000000	+1.00000e+013
15873e-166734	Exponent too large: -166730 ... Issued from stage 2 of pipelin ... Running "spec a: word 1 . prin Processing item number 2: print a

The e character specifies the beginning of the exponent field. Even though case is ignored syntactically within a picture, it is respected in the character to be inserted to signify the beginning of the exponent. The digits of the exponent follow simplified rules for formatting because the exponent is an integer.

In this picture, the number is printed with a leading sign, one digit before the decimal point, five digits decimal fraction, the exponent sign, and three digits exponent.

The last number contains an exponent that is too large to print using the picture specified; but the number is well within the range you can store in a counter.

You can even format numbers according to Continental European conventions:

<i>Figure 314. European Formatting</i>	
spec a: word 1 . print a picture sss.sss.ss9,v99 1	
0	+0,00
-0	+0,00
.04	+0,04
-123456	-123.456,00
1234567	+1.234.567,00

This picture contains both periods and commas. Thus, the v is used to specify the units position explicitly, because the periods are not marking the units position; they mark millions and thousands, respectively. You can also see the way the punctuation characters are suppressed just like the drifting sign. Notice that zero is considered positive.

Boolean Operators

Every *journeyman plumber* knows how to write a multistream pipeline that puts an indication of the equality of words 1 and 2 into column 1. But with *spec*, this can be done much more simply:

<i>Figure 315. Mark Differences</i>	
<pre>spec a: word 1 . b: word 2 . print a==b picture 9 1 1-* 3</pre>	
<pre>1 1 1 1.0 1e1 10 abc def 2 3</pre>	<pre>1 1 1 0 1 1.0 0 1e1 10 0 abc def 0 2 3</pre>

The result of a relational operator is a number, which is zero for failure and one for success. This result is inserted into column one of the output record using a picture that contains a single digit (picture 9). Thus, if the result of the comparison is true, a single digit 1 is placed in column 1 of the output record; otherwise, a single digit 0 is placed in column 1.

You can see that the two equal signs mean that the comparison is strict, as defined for REXX.

To compare a field against a character constant:

<i>Figure 316. Mark Lines that Contain a Literal</i>	
<pre>spec a: word 1 . print a=="1" picture 9 1 1-* 3</pre>	
<pre>1 1e1 abc 2</pre>	<pre>1 1 0 1e1 0 abc 0 2</pre>

To compare a field against a numeric constant:

Figure 317. Mark Lines that Contain a Number	
<pre>spec a: word 1 . print a=1 picture 9 1 1-* 3</pre>	
<pre>1 1e0 1e1 abc 2</pre>	<pre>1 1 1 1e0 0 1e1 Not a decimal number: X'abc' ... Issued from stage 2 of pipelin ... Running "spec a: word 1 . pri ... Evaluating "a=1" Processing item number 2: print a=</pre>

Note the processing of the second and third lines. They contain numbers that have “exponents”; that is scaling by a power of ten. (This confuses REXX programmers too.)

To perform a numeric comparison between two fields:

Figure 318. Mark Differences	
<pre>spec 1-* 3 a: word 1 . b: word 2 . print a=b picture 9 1</pre>	
<pre>1 1 1 1.0 1e1 10 abc def 2 3</pre>	<pre>1 1 1 1 1 1.0 1 1e1 10 Not a decimal number: X'abc' ... Issued from stage 2 of pipelin ... Running "spec 1-* 3 a: wor ... Evaluating "a=b" Processing item number 4: print a=</pre>

Now you see that using a single equal sign makes the comparison numeric. But unlike in REXX, a numeric field must contain a number; *spec* does not revert to strict comparison when it cannot convert a field to the internal representation of a number. It issues an error message and terminates instead.

:

But there is a datatype function you can use to test the operands:

<i>Figure 319. Mark Differences more Carefully</i>	
<pre>spec 1-* 3 a: word 1 . b: word 2 . if (datatype(a)="NUM" & datatype(b)="NUM") then print a=b picture 9 1 else /**Err***/ 1 endif</pre>	
<pre>1 1 1 1.0 1e1 10 abc def 2 3</pre>	<pre>1 1 1 1 1 1.0 1 1e1 10 **Err** 0 2 3</pre>

datatype returns NUM precisely when the conversion to a numeric value will succeed.

There is even a conditional operator. To find the maximum of two fields as one would do in the C programming language style:

<i>Figure 320. Display Maximum</i>	
<pre>spec a: word 1 . b: word 2 . print (a>b ? a : b) 1</pre>	
<pre>1 2 3 1</pre>	<pre>2 3</pre>

The *conditional operator* first evaluates the expression before the question mark. When the result of this expression is not zero, the expression between the question mark and the colon is evaluated and the expression after the colon is ignored. Likewise, when the result is zero, the expression between the question mark and the colon is ignored and the expression after the colon is evaluated. Thus, in this example, field a is tested for being greater than field b. If the result of that test is true, the result of the conditional expression is the value of field a; otherwise, the result is the value of field b.

The expression above is enclosed in parentheses; this allows the use of blanks to make it more readable.

Conditional Processing

You can test the value in a counter or an input field and issue or ignore specification items, depending on the outcome. To mark with an equal sign in column 1 all records where the first two words are equal:

<i>Figure 321. Mark Equality</i>	
<pre>spec a: word 1 3 b: word 2 6 if a==b then /=/ 1 endif</pre>	
<pre>1 2 1 1.0 1 1</pre>	<pre>1 2 1 1.0 = 1 1</pre>

You could not have done this with a conditional expression.

You can supply specification items to be issued as an alternative:

<i>Figure 322. Mark Equality</i>	
<pre>spec a: word 1 3 b: word 2 6 if a==b then /=/ 1 else /-/ 1 endif</pre>	
<pre>1 2 1 1.0 1 1</pre>	<pre>- 1 2 - 1 1.0 = 1 1</pre>

And you can test n ways:

<i>Figure 323. Show Relationship</i>	
<pre>spec a: word 1 1 b: word 2 . if a<b then /</ nextword elseif a>b then />/ nextword else /=/ nextword endif id b nextword</pre>	
<pre>1 2 1 1.0 2 1</pre>	<pre>1 < 2 1 = 1.0 2 > 1</pre>

The first word is copied to the beginning of the output record; the relation that holds is then inserted; and finally, the second word is inserted into the output record. Rather than specifying the input range once again, `id` is used to refer back to the item that defined the field.

You can even iterate. To reverse every second word of the input line:

<i>Figure 324. Iterating Over the Record</i>	
<pre>spec a: 1-* . set #0:=words(a) set #1:=0 while ((#1+=1) <= #0) do if #1//2 then print reverse(word(a, #1)) nw else print word(a, #1) nw endif done</pre>	
<pre>The time has come the Walrus said</pre>	<pre>ehT time sah come eht Walrus dias</pre>

Apart from showing how to write a while group, the example shows an important concept. It makes sure the the counter controlling iteration is always incremented. It does so by starting with a value that is one tick less than the first index wanted and then increment it in the expression that determines when to stop.

If you do it in other ways, you might forget to iterate and then *spec* will go on until the cows come home (this is a technical term meaning forever) and you will be forced to use HX to stop the show, which is a rather hamfisted way of doing so.

You might be tempted to write:

```
while (#1+=1 <= #0) do
```

But then it would loop forever because += is so low in the precedence hierarchy of operators that the increment is 1<=#0 which is always one when there is one or more words in the input line; not what you should want.

It is recommended to use parentheses around the assignment expression to explicitly state that the result is used further in the expression. *spec* issues nuisance warnings in some cases when it detects an operator to the left of the counter being updated.

As shown above, you can nest IF and WHILE constructs; and you can nest any combination. The depth is limited to 16.

The Second Reading Station

After each cycle, *spec* loads the record on the primary input stream into a buffer that is called the *second reading station*, or “second reading” for short.

You can treat this buffer as an additional input stream, which is selected by SELECT SECOND.

Using this, you can combine fields from two adjacent input records without using the READ or READSTOP specification items. Thus, you can construct the output record by intermixing fields from the two records:

<i>Figure 325. Mixing Records</i>	
<pre>spec word 1 1 select second word 1 nextword select first word 2 nextword select second word 2 nextword</pre>	
<pre>first record second line last one</pre>	<pre>first record second first line record last second one line last one</pre>

SELECT FIRST is a convenience for SELECT 0; it selects the primary input stream as the source for the following specification item.

The second reading contains a null record while the first record is being processed. This cycle is called the *runin* cycle.

Likewise, *spec* runs an additional cycle when it reaches end-of-file. This cycle is called the *runout* cycle. The input streams are assumed to contain null records during the runout cycle, but the second reading station still contains the last record.

The ALU supports built-in functions that return true while *spec* is taking a runin or a runout cycle. `first()` is true during the runin cycle; `eof()` is true during the runout cycle.

The runin cycle is skipped if the second reading is the only input stream used. The first record is then loaded directly into the second reading. The runout cycle is skipped when the second reading is not used and no EOF specification item is issued.

Not all specification items in the list are issued during runin and runout cycles. The rules are somewhat arcane; refer to the reference if you are mixing SELECT FIRST and SELECT SECOND.

: The example in Figure 326 on page 196 processes data from the first reading only. This
 : is appropriate for titles or similar that come before the run of records that has a particular
 : key. Use the second reading station when you wish to compute subtotals.

Control Breaks

Field identifiers have other uses than to supply numeric data for computations.

spec can compare a field in two adjacent records and issue specification items only when the fields do not contain the same data. A field identifier defines a field to be compared between adjacent records.

The input file is usually sorted on a key field before being passed to the *spec* stage that generates a report. A *control break* means that the key has changed between two adjacent records.

Suppressing Repetitions

For example, you can suppress the contents of the first five columns in the output record when they are the same as in the previous record:

Figure 326. Avoid Repetition	
<pre>spec a: 1.5 . 6-* 7 break a id a 1</pre>	
<pre>spoonR Spoons, red spoonY Spoons, yellow fork R Forks, red fork Y Forks, yellow</pre>	<pre>spoon R Spoons, red Y Spoons, yellow fork R Forks, red Y Forks, yellow</pre>

The first specification item has a *field identifier* (a) associated with it. The field itself covers the first five columns of the input record. The placement is a period, which means that the item has no effect on the output record. Because the field identifier is used in a subsequent BREAK item, the contents of the field are compared with the contents of the same field in the previous input record, which has quietly been squirrelled away in a buffer for this purpose (the second reading). A break on level a is established when the two fields are not identical. Because there is no previous record when the first record is processed (the previous record is considered null), a break is established on the first cycle.

The second specification item copies the remainder of the input record to the output record, inserting blanks in columns 1-6.

The third item tests if a break is established for field a. If no break is established, the remainder of the specification list is ignored.

The fourth item inserts the key (the contents of field a) into the first five positions of the record. Because this specification item is issued only when a break is established, subsequent output records for this key contain blanks in the first five columns.

Generating Title Records

You can also use WRITE to generate a separate title record, but you must write the title before you build the detail output record, because you can build only one record at a time:

<i>Figure 327. Add Title</i>	
<pre>spec a: 1.5 . if break(a) then /Part number/ 1 id a nextword write endif 6-* 7</pre>	
<pre>spoonR Spoons, red spoonY Spoons, yellow fork R Forks, red fork Y Forks, yellow</pre>	<pre>Part number spoon R Spoons, red Y Spoons, yellow Part number fork R Forks, red Y Forks, yellow</pre>

The built-in function `break()` returns true if a break is established for the field specified. This is used to generate the title line. Note the use of `WRITE`; without it, the title would be prefixed to the output record that is written at the end of the cycle.

Printing Subtotals

Let columns one and two contain the part number and columns three through five contain the number shipped:

<i>Figure 328. Writing a Subtotal</i>	
<pre>spec select second a: 1.2 1 b: 3.3 5 set #0+=b break a write print #0 picture zzz9 4 set #0:=0</pre>	
<pre>mv002 mv003 wv002 wv001 wv002</pre>	<pre>mv 002 mv 003 5 wv 002 wv 001 wv 002 5</pre>

`SELECT SECOND` is issued first to cause the second reading station to be used as the source of the data. Thus, a control break is active while the last record having a particular key is being processed.

The second specification item identifies the part number with `a` and copies it to the output record. Likewise, the third item identifies the number shipped with `b` and prints that as well.

The fourth specification item accumulates the number of items shipped.

A control break is established when the part number of the following record is different from the one in the current record (strictly, when the content of the field identified by a changes).

The last three specification items are issued only when the break is established. The first of them writes the detail record in the output buffer so that the subtotal can be written as a separate record; the second one prints the subtotal (the contents of the counter); and the last one resets the counter to 0.

A subtotal is also printed at end-of-file, because end-of-file forces a break to be established on all levels; thus, there is a break on level a after the last input record has been processed.

Printing a counter and resetting it are often combined:

<i>Figure 329. Writing a Subtotal</i>	
<pre>spec select second a: 1.2 1 b: 3.3 5 set #0+=b break a write print (#0; #0:=0) picture zzz9 4</pre>	
01002	01 002
01003	01 003
02002	5
02001	02 002
02002	02 001
	02 002
	5

In Figure 329, the discard operator (the semicolon) is used to reset the counter after its contents have been fetched for printing. The semicolon operator first evaluates its left hand operand, which is simply the contents of the counter; this becomes the result of the discard operator. It then evaluates the right hand operand and discards that result. Thus, conceptually at least, the contents of counter 0 are moved to the output record before the counter is reset to zero.

As you would expect, the semicolon operator has the lowest precedence of all operators.

Adding a grand total at end-of-file, involves accumulating it and printing it:

<i>Figure 330. Writing a Subtotal with Grand Total</i>	
<pre>spec select second a: 1.2 1 b: 3.3 5 set #0+=b break a write print #0 picture zzz9 4 set (#1+=#0; #0:=0) eof write print #1 picture zzz9 10</pre>	
<pre>01002 01003 02002 02001 02002</pre>	<pre>01 002 01 003 5 02 002 02 001 02 002 5 10</pre>

After each subtotal is printed, it is added into counter 1 to accumulate the grand total.

Printing the grand total is just like printing the subtotal, except that it is done only once.

You can also print the subtotals before you load the data from the first reading station, but now you must suppress printing during the very first break. You must also print the subtotal for the last batch during the runout cycle:

<i>Figure 331. Writing a Subtotal without Second Reading</i>	
<pre> spec a: 1.2 . if (break(a) & ~first()) then print #0 picture zzz9 4 set (#1+=#0; #0:=0) write endif if break(a) then id a 1 endif b: 3.3 5 set #0+=b eof print #0 picture zzz9 4 print #1+#0 picture zzz9 10 </pre>	
<pre> 01002 01003 02002 02001 02002 </pre>	<pre> 01 002 003 5 02 002 001 002 5 10 </pre>

The second specification item tests for a break on level a, except for the break during the runin cycle. If such a break occurs, the subtotal is written in a separate record and the counter reset.

The second IF test unconditionally for a break on level a, which ensures that the first record contains the part number of the first part.

Then the count is printed, as before.

Because the second reading station has not been selected, the runout cycle starts at the EOF specification item. Thus, to print the last subtotal, the specification item to print it is repeated here.

You can see that it was much easier to control totals by processing data from the second reading station, as was done in Figure 330 on page 199.

Break Hierarchies

Control breaks are often hierarchical. When you are generating an invoice, you might wish to group detail records for individual part numbers together and compute subtotals for each. Of course, you would also want to print an invoice total for each customer, and no doubt some grand total at the end.

To do this, you will need to define several types of control breaks; in this case, at least one for part numbers and one for customer numbers.

Note also that when the customer number changes, a control break should be generated for the part number first, even if the part number is unchanged. To support this, control breaks are ordered in a hierarchy, which has a at the lowest level and Z at the highest level.

Here is an example of an utterly simplistic invoicing application. It is an elaboration on the subtotalling examples above. The part number is in columns one and two; the customer number in columns three and four; and the number of items shipped is in columns five through seven:

<i>Figure 332. Simple Invoicing</i>	
<pre>spec select second a: 1.2 1 b: 3.2 4 c: 5.3 8 set #0+=c break a write print (#0; #1+=#0; #0:=0) picture zzz9 7 /total this part/ nextword break b write print (#1; #2+=#1; #1:=0) picture zzz9 7 /total this customer/ nextword eof write print #2 picture zzz9 7 /grand total/ nextword</pre>	
<pre>0101002 0101003 0201002 0202001 0202002</pre>	<pre>01 01 002 01 01 003 5 total this part 02 01 002 2 total this part 7 total this customer 02 02 001 02 02 002 3 total this part 3 total this customer 10 grand total</pre>

SELECT SECOND selects the data source as being the record at the second reading station. Thus, when testing for break the previous record is compared to the current one. It is easier if you readjust your focus to be the record from where data come; then a control break means that the current record is the last one for that particular key.

When *spec* Establishes a Break

Because the READ and READSTOP specification items introduce new data, and also to avoid unnecessary computation, control breaks are established only as needed.

The break() functions return a numeric value that can be printed:

<i>Figure 333. Testing Breaks</i>	
<pre>spec select second a: 1 1 b: 2 2 print break(a) picture 9 nextword print break(b) picture 9 next print eof() picture 9 next</pre>	
xx	xx 110
xy	xy 100
yy	yy 110
yx	yx 000
yx	yx 111

Break level a is associated with the first column and break level “b” is associated with the second one. The result of the break() functions for the two identified fields are printed after the fields have been moved to the output record. The value of the eof() function is also printed.

The function results are printed in the order of the break hierarchy; you can see that the break on b when the first record is at the second reading station forces a break for a as well, even though the first column is unchanged.

But a break is not established until the specification item that defines the associated identifier is issued. Printing the function results after the specification item loading the a field has been issued uncovers some possibly surprising behaviour:

<i>Figure 334. Testing Breaks</i>	
<pre>spec select second 1-* 1 a: 1.1 . /after a:/ 4 print break(a) picture 9 nextword print eof() picture 9 nextword write b: 2.1 . /after b:/ 4 print (break(a)*100 +break(b)*10 +eof()) picture 999 nextword</pre>	
aa	aa after a: 0 0
ab	after b: 110
	ab after a: 1 1
	after b: 111

It is an error to refer to a break level that has not been identified with an input field. Therefore, when the first set of function results is printed, the column for break level b is left blank.

Look at the output for the first record. No break level is established when the specification item identified by a is issued, because the second record also contains a in the first column.

But when the specification item identified by b is issued, a break is established at that level and this forces the break at all lower levels to be established as well.

On the runout cycle, in contrast, the maximum break level is established at the beginning of the cycle.

When you have two separate fields and you wish to issue specification items when either of the fields break, you might be tempted to test only the break level of lower rank, having the expectation that a break of higher rank will force the break on the lower level:

<i>Figure 335. Testing Breaks</i>	
<pre>spec select second a: 1 1 b: 2 2 break a /break/ nextword</pre>	
aa aa ab	aa aa ab break

But this is a mistake, because unless you test the break level, it is not treated as a break level. To issue some specification items when either a or b breaks:

<i>Figure 336. Testing Breaks</i>	
<pre>spec select second a: 1 1 b: 2 2 break b break a /break/ nextword</pre>	
aa aa ab	aa aa break ab break

Thus, even though the first BREAK items looks redundant, it is not. The order is important; you must test the higher rank(s) first.

: Suppressing Detail Printing

: PRINTONLY also supports a letter, which specifies the break level that must be established
 : before an output record is written. The contents of the output buffer is normally discarded
 : when the break level is insufficient, but it is kept if you specify KEEP with PRINTONLY.

: This example shows the use with EOF. The first word of each input record specifies the
 : output column of the remaining words in the record.

spec Tutorial

<i>Figure 337. Keeping the Output Record</i>	
spec printonly eof keep a: word 1 . word 2-* (a)	
7 seven 1 one 13 thirteen	

Had KEEP been omitted, you would see the last record only:

<i>Figure 338. Not Keeping the Output Record</i>	
spec printonly eof a: word 1 . word 2-* (a)	
7 seven 1 one 13 thirteen	

Driving *spec* with Due Care and Attention

If you use the divide operator, you must consider the possibility of dividing by zero:

<i>Figure 339. Dividing by Zero</i>	
spec a: word 1 . set (#0+=a; #1+=1) eof print #0/#1 1	
	Divisor is zero ... Issued from stage 2 of pipelin ... Running "spec a: word 1 . set ... Evaluating "#0/#1" Processing item number 4: print #0

In this example, there are no input records; thus, both counters contain zero at end-of-file. *spec* does not treat zero divided by zero in any special way; it reports the error rather than risking the potential divide exception. You can prevent an error in such a case:

Figure 340. Dividing by Zero

```
spec
a: word 1 .
  set (#0+=a; #1+=1)
eof
if #1>0 then
  print #0/#1 1
else
  /No input records./ 1
endif
```

```
No input records.
```

Examples

Page Formatter

When printing a file, you might wish to add headings and page breaks in the same way accounting machines used to do. You can write a REXX filter for this or use *specs*.

The example in Figure 341 on page 206 generates an output file that contains ASA carriage control in the first column. The number 1 means that the record should be printed at the top of the next page; a blank means that the record should be printed on the next line.

The arguments specify the page size in lines and columns.

Figure 341. Making Page Breaks

```

/* Print pages with 60 lines and a heading                               */
Signal on novalue

parse arg pl pw
If pl=''
  Then pl=60
If pw=''
  Then pw=80

'callpipe (name PRTPAGE)',
'|*:',
'|specs',
  'if #0=0 then',
  '  ?1'date()'?' 1',
  '  ?Page?' pw-10,
  '  print #1+=1' pw-5'.5 right',
  '  write',
  '  / / 1',
  '  write',
  '  set #0:=3',
  'endif',
  '1-* 2',
  'set (#0 := #0>='pl '?' 0 : #0+1)',
'|*:'
exit RC
/* Top of page? */
/* Title */
/* And literal */
/* Page number from #1 */
/* Write title line */
/* Blank line */
/* Write second line */
/* First copied line is line 3 */
/* The line with one space */
/* Test page overflow */

```

Counter 0 is used to keep track of the current line number on the output page; counter 1 is used for the current page number. When the line number is 0, a page header is written and the page number is incremented. Then a blank line is written, and the line number is set to 3. Note that counter 0 is initialised to zero, which means that a heading is included in the first page.

After each input record has been written to the output, the line number is tested for being greater than or equal to the page size. If it is, it is reset to zero so that a heading is generated in the next cycle; otherwise it is incremented by 1.

Here it is run using a small page size:

Figure 342. Using PRTPAGE REXX

prtpage 4 30			
Line 1	129 Apr 2020	Page	1
Line 2			
Line 3	Line 1		
Line 4	Line 2		
Line 5	129 Apr 2020	Page	2
	Line 3		
	Line 4		
	129 Apr 2020	Page	3
	Line 5		

: **And Finally**

: Before leaving this tutorial we remove the definition of the structures we defined in
 : Figure 277 on page 170.

```

: pipe literal str|structure delete thread
: ▶Ready;

```

: *Figure 343. Deleting Structure Definitions*

Chapter 17. Rita, the *CMS Pipelines* Runtime Profiler

Rita reports on the CPU usage of a pipeline set by stage and pipeline specification. Rita also reports the largest amount of virtual storage used by each stage for work areas and buffers.

Rita comes with CMS on the examples disk, usually MAINT 193.

To invoke Rita, change your PIPE command to RITA. RITA invokes the PIPE command to run the pipeline with options to capture timing information and a stage to reduce this information.

Rita displays CPU usage in milliseconds for each stage and pipeline, both inclusive of and exclusive of the time used by subroutine pipelines invoked by the stage or pipeline.

Rita writes a summary on the console and detailed information to a disk file. The file name of Rita's output file is the first eight characters of the option NAME specified on the RITA command. The file type is of the form RITAnnn, where nnn is the first unused sequence number.

For a more detailed discussion and many examples of using Rita to tune pipelines, see Melinda Varian's *Streamlining Your Pipelines* on the *CMS Pipelines* home page. Refer to "Additional Information, Download Site" on page xx for additional pointers.

If you are too busy to read this extremely informative paper, beware of this:

- If a stage (it would have been written by a user) goes into a wait state outside of the control of *CMS Pipelines*, the wait time is counted as CPU time.
- Pipelines added with ADDPIPE are not represented in the inclusive number for any stage.
- The numbers displayed by Rita do not include the CPU usage of other pipeline sets started with PIPE commands or *runpipe* stages from within the pipeline set Rita is measuring.
- Rita produces the best results when all CALLPIPES and ADDPIPES have the option NAME specified and when any specifications that have the same name are also identical in the stages invoked.
- NUCXLOAD RITA before running a pipeline that contains a *ldrtbls* stage. This avoids Rita interfering with the loader tables.
- Rita is likely to add less overhead when RITA REXX is compiled.

While the results from Rita will be indicative of the relative performance of various stages and subroutine pipelines, Rita comes at price:

- Rita enables the message level to gather accounting data. This, in turn, causes the pipeline dispatcher to take a longer path than it would take otherwise.
- Rita issues the pipeline specification through *runpipe* and processes the output to extract pipeline accounting messages. This may add an overhead of 5% or more.

If the application you are timing fails because of the additional execution time, you may try to:

- Run the pipeline through *runpipe* MSGLEVEL X2001 and save the output to disk. Then process the file through *rita*; that is, RITA REXX.
- Turn on console SPOOL, issue PIPMOD MSGLEVEL 8193 to enable pipeline accounting, and run the pipeline normally. Inspect the console SPOOL for message 177.

Example

Invocation and summary followed by detailed output truncated on the right:

```

rita literal abc|append literal def|hole
▶ CPU Utilization by Pipeline Specification          28 Nov 2006 16:56:46
▶
▶ 0.003 ( 0.003) ms total in "Append/Preface" (1 invocation)
▶ 0.013 ( 0.010) ms total in "NoName001" (1 invocation)
▶
▶ 3.078 ms total.
▶
▶Detailed output from Rita in UNNAMED RITA001.
▶Ready; T=0.21/0.25 16:56:46
  pipe < unnamed rita001|cons
▶CPU Utilization by Pipeline Specification          from: 28 Nov 2006 16:56:46
▶                                                    to: 28 Nov 2006 16:56:46
▶
▶CPU utilization of pipeline specification "Append/Preface":
▶ 0.003 ( 0.003) ms ( <1K) in stage 1 of pipeline 1: literal de
▶ 0.003 ( 0.003) ms total in "Append/Preface" (1 invocation) <=====
▶
▶CPU utilization of pipeline specification "NoName001":
▶ 0.003 ( 0.003) ms ( <1K) in stage 1 of pipeline 1: literal ab
▶ 0.006 ( 0.003) ms ( <1K) in stage 2 of pipeline 1: append lit
▶ 0.004 ( 0.004) ms ( <1K) in stage 3 of pipeline 1: hole
▶ 0.013 ( 0.010) ms total in "NoName001" (1 invocation) <=====
▶
▶ 0.013 ms attributed to stages; no virtual I/O.
▶ 2 pipeline specifications used (4 stages).
▶ 2 pipeline specifications issued.
▶
▶ 0.005 ms in general overhead.
▶ 0.018 ms in scanner.
▶ 0.006 ms in commands.
▶ 1.024 ms in dispatcher.
▶ 0.000 ms in hunt.
▶ 2.012 ms in accounting overhead.
▶
▶ 3.078 ms total.
▶Ready; T=0.01/0.08 16:58:57

```

In this contorted example, dispatcher and accounting overhead completely overshadows any other CPU use. Note also that while Rita discovers 3 milliseconds CPU consumption, the actual CPU time for the entire command is 210 milliseconds.

Chapter 18. Using VM Data Spaces with *CMS Pipelines*

The next chapter describes some *CMS Pipelines* built-in programs that have been enabled for data space access. *TSO Pipelines* cannot create shared data spaces, as such operations require the task to be authorized.

This chapter describes how to combine *CMS Pipelines* built-in programs to manage data spaces, address list element tokens, and memory mapped minidisks. It contains a complete terminal session as examples, including supporting commands that are not directly related to data spaces. If you choose try the session yourself, you should perform the steps in the same sequence as shown here and in one virtual machine and pay attention to the contents written to files where an ASIT is saved to disk for later reference.

CMS Pipelines does not expose the entire repertoire of CP macros that are available and it also makes a few simplifying assumptions.

Your virtual machine must be in XC mode to use the data space support and you must have been given privileges in the user directory entry for your virtual machine if you wish to create data spaces or share them, because by creating a data space you increase your virtual machine's footprint. This line will allow you to create up to ten address spaces of a maximum aggregate size of one gigabytes; further, you are allowed to share the data spaces you create:

```
xconfig addrspace maxnumber 10 tosize 1g share
```

The CP support for VM data spaces is described in *z/VM CP Programming Services*, SC24-6272. You may also find the online help files useful when developing pipelines that use data spaces. Issue the CMS command HELP VMDS MENU to display a menu of the CP macros in support of data spaces.

CMS Pipelines provides interfaces to most of the CP macros with *adrspace*, *alserv*, and *mapmdisk*. In addition, *diskid* supports reserved minidisks.

Terminology

Your virtual machine's real storage is formally called the *host-primary address space* of your virtual machine. With appropriate privileges, you can also create data spaces, which are separate sets of pages that can contain data, but from where no instruction can execute. The virtual machine real storage and data spaces are collectively referred to as *address spaces*.

An address space has a name; it is also represented by an *address space identification token*, an ASIT, which is eight bytes.

The name of your real storage is the reserved word "BASE". You give a data space a name when you create it; the name is up to twenty-four characters made up from the twenty-six English letters, digits, or any of the special characters # \$ @ _ - (number sign, dollar sign, at sign, underscore, and hyphen). Note that the first three special characters are national use; your terminal and keyboard may display these differently. Data space names are upper case, but *CMS Pipelines* translates them automatically, so you can specify them in whatever case you like. The combination of user name and data space name must be unique; that is, a virtual machine can have only one data space by a particular name at any time.

An ASIT that is for the real storage of a virtual machine is called a *virtual configuration identification token* (VCIT); it identifies the address space uniquely within the IPL of the VM system; that is, ASITs are not reused until the system has been shut down. The VCIT identifies the virtual machine uniquely in standard CMS.

You can discover the ASIT by creating an address space or by querying it.

To use a data space you must obtain an *access list entry token* (ALET) for the address space. This value is loaded into an *access register* by *CMS Pipelines* to identify the address space you wish to reference. If you are writing Assembler programs, you would then switch to access register mode to access the data space and switch back to primary space mode when you are done.

The contents of a data space is either something you put there or it is mapped to a mini-disk. The contents lasts until the data space is destroyed or you IPL the virtual machine. As CMS does not know of data spaces, end of command has no effect on a data space.

Querying an Address Space

You already have an address space, namely the real storage of your virtual machine. This address space has the reserved name "BASE"; pass it to *adrspc* QUERY. Figure 344 shows how to determine the ASIT of the virtual machine's real storage and the number of pages in it.

Figure 344. Determining the Base ASIT.

```
pipe literal base | adrspc query | spec 1-8 c2x 1 9-* c2x nw | console
▶00F2A9C000000004 00004000
▶Ready;
```

The contents of the ASIT is not specified; it is just a handle, but you are assured that the value is unique for the duration of the IPL of VM.

However, if you watch the ASIT of your real storage you will soon note that the lower word is incremented as you IPL your virtual machine and as you create data spaces.

Figure 345. Querying an Extinct Data Space

```
pipe literal nixen bixen | split | adrspc query | cons
▶FPLASP1527E Address space NIXEN is not available for user
▶FPLMSG003I ... Issued from stage 3 of pipeline 1
▶FPLMSG001I ... Running "adrspc query"
▶FPLASP1527E Address space BIXEN is not available for user
▶FPLMSG003I ... Issued from stage 3 of pipeline 1
▶FPLMSG001I ... Running "adrspc query"
▶Ready;
```

This attracts a warning, but does not terminate *adrspc*. In fact, *adrspc* acts like a selection stage; it passes the name of the unknown data space to its secondary output stream, when it is defined.

VM Data spaces

Figure 346. Querying an Extinct Data Space

```
pipe (end ?) literal nixen bixen | split | a: adrspace query | ...
... > good names a ? a: | insert /dunno: / | console
▶dunno: NIXEN
▶dunno: BIXEN
▶Ready;
```

Finding one's VCIT is done so often that you might write MYASIT REXX as shown in Figure 347.

Figure 347. MYASIT REXX

```
/* Write primary ASIT */
Signal on novalue
numeric digits 12

'callpipe (end \ name MYASIT.REXX:6)',
  '\literal BASE',
  '| adrspace query',
  '| chop 8', /* Drop size */
  '|*:'
exit RC
```

Displaying data in hex is also done a lot; C2X REXX hides the complexity of formatting four bytes at a time (this is not a trivial demonstration of the capabilities of *spec*).

Figure 348. C2X REXX

```
/* Convert char to hex */
Signal on novalue
numeric digits 12

parse arg as
If as\=''
Then 'issuemsg 112 FPLC2X x'c2x(as)
else
'callpipe (end \ name X2C.REXX:6)',
  '\*:',
  '|spec set #0:=length(record());#1:=-3',
  '| while (#1+=4)<=#0 do',
  '| if (#1>1 & #1//16=1)',
  '| then / / n endif',
  '| print c2x(record(#1, 4)) nw',
  '| done',
  '|*:'
exit RC
```

Accessing the Contents of a Data Space

The *storage* built-in program can access the contents of a data space. Figure 349 on page 213 shows how to obtain an ALET, use it to display data in your virtual storage, and finally dispose of the ALET.


```

:
: Figure 349. Displaying Data in a Data space
:
: pipe myasit | alserv add | > base alet a | c2x | console
: ▶01000002
: ▶Ready;
:
: pipe storage alet 01000002 200 32 | console
: ▶z/VM V6.4.0 2019-07-24 16:40
: ▶Ready;
:
: pipe < base alet | alserv remove
: ▶Ready;
:

```

This example is contrived because you could obtain the same information directly without specifying ALET, but it still shows the mechanics.

You can also access someone else’s data spaces, if that virtual machine has granted you permission and you know the ASIT. Either the owner of the data space has left the ASIT in a prearranged place, for example in SFS or the address space name is well known, in which case you can use *adrspac* QUERY to discover it for yourself.

Creating a Data Space

It is quite straightforward to create a data space. Give it a name, specify how large, and assign a storage key. Pass this information on the input to *adrspac* CREATE.

```

:
: Figure 350. Creating a Data space
:
: pipe literal ds1 2 e0 | adrspac create | > ds1 asit a | ...
: ... spec 9-* c2d 1 | console
: ▶ 256
: ▶Ready;
:

```

The output is the ASIT of the data space. Note that the size has been rounded up to the nearest megabyte segment. It is a good idea to save the ASIT either in a disk file or a REXX variable. As there is no REXX environment active, we store it in a file here. (But you can always discover it by *adrspac* QUERY as long as you remember the name of the data space.)

To access the contents of the data space we need an ALET:

VM Data spaces

Figure 351. Creating a Data space

```
pipe < ds1 asit a | spec 1-8 c2x 1 9-* c2x nw | > ds1 asitx a | console
▶00F2AAC000000001 00000100
▶Ready;

pipe < ds1 asit | alserv add write | > ds1 alet a | c2x | console
▶01000002
▶Ready;

pipe storage alet 2 0 16 | spec 1-* c2x 1 | console
▶00000000000000000000000000000000
▶Ready;

pipe storage alet 0 0 16 | spec 1-* c2x 1 | console
▶03EC200083A480CA03A0BD2883AB23F6
▶Ready;
```

This creates a file that will be needed in Figure 358 on page 217.

Note that we now have a megabyte of shiny new zero bits available by using ALET 2. Contrast it with the contents of real storage as shown in the last command. (ALET 0 is reserved and always refers to the primary space of the virtual machine; ALET 1 is not valid on CMS; it is used to reference the secondary address space on z/OS.)

You can omit the leading X'01' of an ALET on CMS; *CMS Pipelines* supplies it for you as that is the only format that CP supports.

The ALET is a number between 2 and the maximum number of ALETS allowed for your virtual machine, up to 1023, which is the limit in the hardware architecture. Their numbers are predictable, being the smallest unassigned number.

Let us put something into the data space and even in the first two pages for good measure:

Figure 352. Loading Data into a Data Space

```
pipe literal Killroy was here | pad 32 | storage alet 2 0 32 e0
▶Ready;

pipe literal Killroy was also here | pad 32 | storage alet 2 1000 32 e0
▶Ready;

pipe storage alet 2 0 16 | console
▶Killroy was here
▶Ready;
```

And you are undoubtedly not surprised to see the data staying in the data space.

Sharing Address Spaces

Address spaces can be shared either read only or read/write, but you must yourself implement any locking protocol to manage concurrent update by multiple virtual machines.

Pass the ASIT of an address space that you own to *adrspace* PERMIT.

Figure 353. Sharing an Address Space

```
pipe myasit | adrspace permit user operator | c2x | console
▶00F2A9C0 00000004
▶Ready;
```

Here the user OPERATOR is given read only access to the real storage of your virtual machine.

Use *adrspace ISOLATE* to stop sharing an address space. This is all or nothing at all: all permissions on the data space are lost; you cannot remove permissions for one user but leave those for others.

Figure 354. Removing Permissions to an Address Space

```
pipe myasit | adrspace isolate | c2x | console
▶00F2A9C0 00000004
▶Ready;
```

That said, sharing of address spaces is not as easily done as it might seem:

- The grantee must discover the ASIT, which can become rather complicated, and add an ALET to its access list.
- VM has no facility to grant public access to a data space; permissions must be granted individually.
- All permissions to a data space are dropped when it is isolated; there is no facility to drop a particular permission.
- The virtual machine that is granted permission must be logged on.
- IPL or reset of a virtual machine drops all permissions granted to it previously.
- IPL or reset of the permitting virtual machine deletes all data spaces and clears all permissions granted on them.
- An ALET that you have obtained for a data space in another virtual machine may thus go stale at any time. This is reflected by an addressing capability exception, program interrupt code X'136', from which *CMS Pipelines* cannot recover. Stages that use an ALET are, however, able to determine its validity while validating operands.

Thus, to set up a service virtual machine to maintain a database in a shared data space, you will also need to implement some kind of protocol to enable the server to authorize clients. Refer to “Example Server Application” on page 158 for an example.

Using Mapped Minidisks

You can map the contents of a minidisk into a data space you own and you can effectively save the contents of a data space to a minidisk; this depends on the way you define it.

Mapped minidisks are used by DB/2 for VM to access the database directly as virtual storage. While you can map any disk that is formatted with 4K blocks, maintaining the file system structures or even just accessing the contents of files is not trivial, but a mapped minidisk would be appropriate for a disk repair kit.

VM Data spaces

Let us get a temporary disk to play with and format it. It must be formatted with 4K blocks, but that is the default for 3390s, so we need not specify that option.

Figure 355. Creating a Temporary Reserved Minidisk

```
pipe cp DEFINE T3390 102 1 | console
▶DASD 0102 not defined; temp space not available
▶Ready(00091);

pipe literal ds1 1 | split | stack lifo | hole | ...
... append command FORMAT 102 W | console
▶DMSFOR113S Device 102 not attached or invalid device address
▶Ready(00100);
```

The output from `FORMAT` is truncated in formatting. The somewhat strange way of providing responses to the prompts from CMS command `FORMAT` allows the sample to be run automatically while this book is formatted.

Figure 356. Creating a Temporary Reserved Minidisk

```
pipe strliteral /1/ | stack lifo | hole | ...
... append command RESERVE ds1 reserved W | console
▶Filemode W not accessed
▶Ready(00036);

pipe state * * w | console
▶Ready(00036);

pipe diskid 102 | spec 1.2 c2x 1 3.2 c2d nw 5.4 c2d nw | console
▶Device 102 is not attached
▶... Issued from stage 1 of pipeline 1
▶... Running "diskid 102"
▶Ready(01538);
```

(`RESERVE` does not like a trailing blank in the response to the prompt.)

Strictly speaking, we could omit reserving the minidisk and use all of it, but reserving the disk prevents trouble if it should ever be accessed. There is no need to access the minidisk; you could read the contents of the disk with `trackread` rather than `mdiskblk`, which does require the minidisk to be accessed.

We now have 172 blocks to play with at offset 8 from the beginning of disk 102. To store the data space into this file we must first define a minidisk pool; in this case the pool will contain one extent only, the temporary disk.

Figure 357. Creating a Minidisk Pool

```
pipe literal 102 8 172 | mapmdisk identify | console
▶Return code 8 on ADRSPACE/ALSERV/MAPMDISK diagnose
▶... Issued from stage 2 of pipeline 1
▶... Running "mapmdisk identify"
▶Ready(01520);
```

This assigns the reserved portion of the minidisk to blocks 0 through 171 of the minidisk pool. The null record indicates that the pool has been defined without error. (Had you for

some reason not passed any extent definitions to *mapmdisk* IDENTIFY, it would not have produced an output record.)

A virtual machine can have only one minidisk pool defined at any time; any existing pool is quietly replaced by the new one.

We are now ready to map the minidisk into the data space that was created in Figure 351 on page 214. We map just the first page of the data space onto the first block of the reserved file. *RETAIN* instructs *mapmdisk* to leave the contents of the data space intact, the default being to use the data on the minidisk.

Figure 358. Mapping a Minidisk Pool into a Data space

```
pipe literal ds1 | adrspace query | spec 1.8 c2x 1 /0 1 0/ nw | ...
... mapmdisk define retain | > ds1 mdmap a
▶No minidisk pool has been defined
▶... Issued from stage 4 of pipeline 1
▶... Running "mapmdisk define retain"
▶Ready(01543);
```

It would appear that the minidisk pool is not referenced once pages have been mapped and that the pool could be redefined while pages are mapped, but this is not documented to be the case.

Having mapped the data space, we save the contents to the minidisk. *mapmdisk* *SAVE* waits while CP performs whatever page out operations are required for changed pages. It writes parameters from the interrupt that marks the completion of the operation.

Figure 359. Saving the Contents of a Data space to a Mapped Minidisk

```
pipe literal ds1 | adrspace query | spec 1.8 c2x 1 /0/ nw | ...
... mapmdisk save | c2x | console
▶00000000 00000000 0100
▶Ready;
```

The output from *mapmdisk* *SAVE* is almost all zeros when the data space has been hardened onto the minidisk. X'01' in byte 9 means that we have received a confirmation interrupt for the save operation; no other values are possible. The leftmost bit of byte 10 indicates the validity of the first eight bytes; the contents are valid when this bit is zero (which is a bit unconventional); the rightmost seven bits of this byte contain the completion status, which should be X'00'. Error codes are described in *z/VM CP Programming Services*, SC24-6272.

You can run multiple *mapmdisk* *SAVE* stages concurrently, for example one for each data space.

Now check the reserved file:

VM Data spaces

Figure 360. Reading from the Reserved Space on a Minidisk

```
pipe < ds1 reserved w | take 1 | chop 32 | console
▶Mode W not available or read only
▶... Issued from stage 1 of pipeline 1
▶... Running "< ds1 reserved w"
▶Ready(00119);

pipe < ds1 reserved w | drop 1 | take 1 | chop 32 | c2x | console
▶Mode W not available or read only
▶... Issued from stage 1 of pipeline 1
▶... Running "< ds1 reserved w"
▶Ready(00119);
```

So the first page was hardened, but the second one was not, as we should expect.

Then let us destroy the mapping of the data space, but keep the data space:

Figure 361. Unmapping a Data space

```
pipe < ds1 mdmap | mapdisk remove
▶File "DS1 MDMAP *" does not exist
▶... Issued from stage 1 of pipeline 1
▶... Running "< ds1 mdmap"
▶Ready(00146);

pipe storage alet 2 0 16 | c2x | console
▶D2899393 9996A840 A681A240 88859985
▶Ready;

pipe storage alet 2 1000 32 | console
▶Killroy was also here
▶Ready;
```

Unmapping a mapped page also discards its contents; unlike when mapping, CP offers no choice this time. Of course, the page that was not mapped retains its contents.

The first page went away, but we can have it back by redefining the mapping.

Figure 362. Restoring a Minidisk Mapping

```

pipe < ds1 mdmap | mapmdisk define
▶File "DS1 MDMAP *" does not exist
▶... Issued from stage 1 of pipeline 1
▶... Running "< ds1 mdmap"
▶Ready(00146);

pipe storage alet 2 0 32 | console
▶Killroy was here
▶Ready;

pipe storage alet 2 1000 32 | console
▶Killroy was also here
▶Ready;

```

Destroying a Data Space

Finally we destroy the sample data space and its ALET.

Figure 363. Destroying a Data space

```

pipe < ds1 asit | adrspace destroy
▶Ready;

pipe < ds1 alet | alserv remove
▶Ready;

pipe command RELEASE W
▶Ready(00036);

pipe cp DETACH 102
▶Ready(00040);

```

As data spaces consume CP resources, a good citizen destroys unneeded data spaces. If you do not, it all goes away in a small puff of white smoke next time you IPL your virtual machine, even permissions you have granted on your base machine storage.

Chapter 19. *CMS Pipelines* Built-in Programs supporting Data Spaces

This chapter shows how to use ALETS with some *CMS Pipelines* built-in programs; it is entirely up to you how you obtain the ALETS involved. On CMS, this would typically be *adrspac* CREATE INITIALISE; some other means must be used on z/OS.

For the examples, we use a single data space, which is created as shown in Figure 364.

Figure 364. Creating a Data space

```
pipe literal sample 2 e0 | adrspac create initialise | > ds1 asit a
▶Ready;
```

Several examples in Chapter 18, “Using VM Data Spaces with *CMS Pipelines*” on page 210 show how to display and store data using STORAGE.

instore and *outstore* also support an ALET operand.

Figure 365. Using *instore* and *outstore*

```
pipe literal Hello! | instore alet 2 | outstore | console
▶Hello!
▶Ready;
```

While the format of the output of *instore* is unspecified, it contains the ALET into which the file is stored. The point to note is that *outstore* determines the ALET from the descriptor record it reads; you need not specify it (you cannot specify it unless *outstore* is first in the pipeline).

When the file is in a data space, *outstore* copies each record into a buffer in the primary space before writing it to its output; thus, other parts of *CMS Pipelines* are not aware of address spaces.

Figure 366. Cleaning Up.

```
pipe < ds1 asit | adrspac destroy | substr 13-* | alserv remove
▶Ready;
```

Part 4. Reference

This part of the book contains reference information.

Chapter 20, “Syntax Notation” explains how to read syntax diagrams.

Chapter 21, “Syntax of a Pipeline Specification Used with PIPE, *runpipe*, ADDPIPE, and CALLPIPE” defines the syntax of the PIPE command, the ADDPIPE pipeline command, the CALLPIPE pipeline command, and the input to the *runpipe* built-in program.

Chapter 22, “Scanning a Pipeline Specification and Running Pipeline Programs” describes how the pipeline specification parser and the pipeline dispatcher go about processing a pipeline specification.

Chapter 23, “Inventory of Built-in Programs” describes the programs that are supplied with *CMS Pipelines*; it describes the syntax of the argument string as well as the operation of the program.

Chapter 24, “*spec* Reference” describes all features of *specs*.

Chapter 25, “Pipeline Commands” describes pipeline commands. Pipeline commands are processed by the default command environment in pipeline filters that are written in REXX.

Chapter 26, “Message Reference” lists *CMS Pipelines* messages in numerical order and explains what they mean.

Chapter 27, “PIPMOD Command (*CMS Pipelines* only)” describes the functions performed by the PIPMOD command (the main pipeline module).

Chapter 28, “Configuring *CMS Pipelines*” describes *CMS Pipelines* configuration variables, which control the actions taken where there has traditionally been a difference between z/VM and the “Pipelines Runtime Library”.

!

Chapter 20. Syntax Notation

This chapter defines the syntax notation used to describe *CMS Pipelines* commands and programs.

Syntax defines valid argument strings for a command. *Semantics* define what the command does when it is issued. For instance, when a program accepts a list of items of some kind as the argument, syntax does not prescribe a limit on the count of items; semantics might require a maximum of 10 entries in such a list.

How to Read a Syntax Diagram

Follow the path of the line from left to right, from top to bottom.

- ▶▶—— The definition of a program or command begins with two arrowheads pointing to the right.
- ▶◀ The definition of a program or command ends with two arrowheads pointing to each other.
- ▶ An arrowhead pointing to the right at the end of a line means that the definition is continued below.
- ▶—— An arrowhead pointing to the right at the beginning of a line means that the definition is continued from above.

A required item is on the main path along the horizontal line.

▶▶—COMMAND—*required argument*—▶◀

A default item is above the main path.

▶▶—COMMAND—default argument—▶◀

An optional item is below the main path.

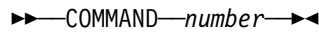
▶▶—COMMAND—optional argument—▶◀

An item is a keyword, a syntax variable, or a reference to a fragment in a syntax definition.

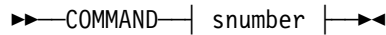
Keywords are shown in a Gothic font with the minimum abbreviation in upper case. When writing the keyword, you must provide at least the minimum abbreviation. Write the keyword in upper case or lower case; write it mIxEd if you like.

▶▶—COMMAND—KEYword—▶◀

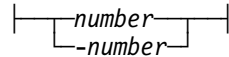
Syntactic variables are shown in lower case slanted type. Provide a number, address, or the name of an object where there is a syntactic variable. Figure 367 on page 224 defines the syntax variables used by *CMS Pipelines*.



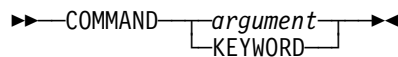
A reference to a fragment of a syntax definition breaks the main path with vertical bars. The fragment is defined later in the diagram.



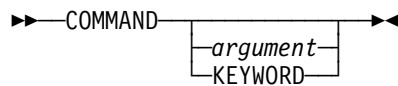
snumber:



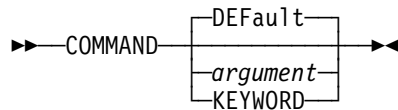
When you must choose between two or more items, they are stacked with the first one on the main path.



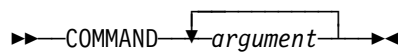
When you can select an item or take none, the choices are stacked below the main path.



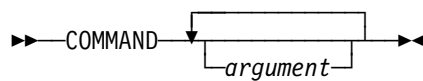
A default is shown above the main path.



An item may be repeated when an arrow returns to the left in front of it. The item is on the main path when you must write it at least once.



The item is below the main path when you may omit it altogether.



Syntactic Variables

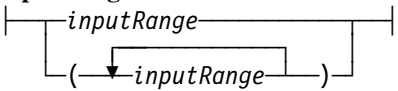
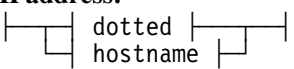
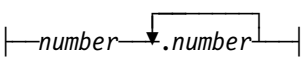
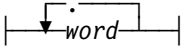
Words in slanted Gothic type beginning with a lower case letter are syntactic variables. Substitute something for the syntactic variable; its name is intended as a mnemonic for the type of information you must substitute.

Syntax

Figure 367 (Page 1 of 5). Syntactic Variables

Name	Examples	Description
<i>bit</i>	0 1	A binary digit, the characters 0 or 1.
<i>blank</i>		Space; the blank character, X'40'.
<i>char</i>	- x	A single character that is not a blank. Any 8-bit value other than X'40' is a <i>char</i> .
<i>delimitedString</i>	/abc/ ,, xf1f2f3 b11000001 str xabx	<p>A delimited character string is written between two occurrences of a delimiter character, as a hexadecimal literal, or as a binary literal. The delimiter cannot be blank and it must not occur within the string. Two adjacent delimiter characters represent the null string. It is suggested that a special character be used as the delimiter, but this is not enforced. However, it is advisable not to use alphanumeric characters, because a future release might add a keyword or a number as a valid option to a built-in program where only a delimited string is valid today.</p> <p>A hexadecimal literal is specified by a leading H or X followed by an even number of hexadecimal digits. A binary literal is specified by a leading B followed by a string of 0 and 1; the number of binary digits must be an integral multiple of eight.</p> <p>The keyword STRING can be used to specify that the delimited string contains a string that is terminated by delimiter characters. This acts as a placeholder so that any non-blank character can be used as the delimiter character. Note that this use of the keyword is in addition to a keyword that is recognized by a built-in program. (Thus, split string string xabcx)</p>
<i>delimiter</i>	/ ,	A single character, which is used to delimit a string. When <i>delimiter</i> is used in a syntax expression, all occurrences of the delimiter refer to the same character.
<i>devaddr</i>	c 00d	A string of hexadecimal digits is also used to express a device address (though its proper name is a device name nowadays). At most eight significant digits are allowed.
<i>digit</i>	5 9	One of the digits 0 through 9.
<i>endChar</i>		The end character; the character declared by the option ENDCHAR. Refer to Chapter 8, "Using Pipeline Options" on page 120.
<i>hex</i>	7 b D	A character selected from the decimal digits 0 through 9, the letters a through f, and the letters A through F.
<i>hexString</i>	00d7	A hexadecimal string consists of one or more hexadecimal digits. No blanks are allowed in such a string. Semantics often require that the string has an even number of characters.

Figure 367 (Page 2 of 5). Syntactic Variables

Name	Examples	Description
<p><i>identifier</i></p>	<p>Struct first_fish m7</p>	<p>This applies to identifiers parsed by <i>structure</i>: Structure and member names (often referred to as identifiers) must begin with a letter in the English alphabet or one of the special characters “@#?!?” (at sign, number sign, dollar sign, exclamation point, question mark, and underscore). The second and subsequent character may also be a digit.</p> <p>Identifiers are case sensitive unless the structure is defined as caseless.</p> <p>Identifiers parsed by <i>polish</i> follow the conventions of the High Level Assembler. That is, exclamation point and question mark are not valid characters in such an identifier. While the parser retains the case of identifiers, the evaluator (which the user must supply) should treat them as caseless.</p>
<p><i>inputRange</i></p>	<p>1-* word 5 1;-1 -18;28 field 4</p>	<p>Refer to “Input Range” on page 228.</p>
<p><i>inputRanges</i></p>	<p>7 1-* (4-* w6) (f3 w7)</p>	<p>A list of input ranges is a single <i>inputRange</i> or a list of input ranges in parentheses. If the keyword WORDSEPARATOR or FIELDSEPARATOR is specified, it remains in effect for subsequent words or fields.</p> <p>inputRanges:</p> 
<p><i>IPaddress</i></p>	<p>9.55.5.13 jph piper.com</p>	<p>An IP address can be expressed in dotted-decimal notation or as a host name, optionally qualified by a domain.</p> <p>Note: The IP address is a single <i>word</i>; you must write IP addresses without embedded blanks.</p> <p>IPaddress:</p>  <p>In the dotted-decimal notation, you are expressing a thirty-two bit integer. While you can specify this number in several ways; the customary notation consists of four integers in the range from 0 to 255, separated by periods. Be sure not to specify leading zeros, as this implies octal notation in some contexts, but not in others.</p> <p>dotted:</p>  <p>The IP address may also be specified by a host name or a hostname followed by a period and a domain name. This usage is experimental on CMS; it requires RXSOCKET Version 2.</p> <p>hostname:</p> 

Syntax

Figure 367 (Page 3 of 5). Syntactic Variables

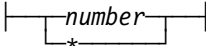
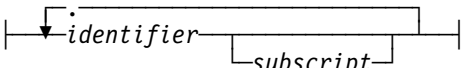
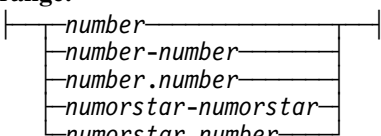
Name	Examples	Description
<i>letter</i>	q	A character in the English alphabet, a through z and A through Z.
<i>number</i>	0012 00 3	A number is a sequence of decimal digits. <i>CMS Pipelines</i> stores numbers as binary fullwords; the largest number supported is 2**31-1 (2147483647). A <i>number</i> is unsigned; that is, zero or positive; semantics often require that a number be positive.
<i>numorstar</i>	17 *	Column numbers are positive integers; the first (leftmost) column is number 1. An asterisk ('*') refers to the first or last column of a record. numorstar: 
<i>octalDigit</i>	5	One of the digits 0 through 7.
<i>qualifier</i>	str str.substr str.sub(4)	A qualifier is a left part of a fully qualified member name. Blanks are not allowed in a qualifier. qualifier: 
<i>quotedString</i>	"a b" "x"y" 'abcd'	A quoted string is written in the REXX fashion. Two consecutive quotes within the string are replaced with a single one.
<i>range</i>	8 1.5 10-12 9-* *.8 9.3 *-*	A range is often used to specify a range of columns in a record or a range of record numbers in a file. It is a single number, the beginning and end of the range with a hyphen ('-') between them, or the beginning number and the count with a period ('.') between them. 10-12 and 10.3 express the same range. No blanks are allowed between the numbers and the delimiters because <i>CMS Pipelines</i> scans for a <i>word</i> before scanning the word for the range. The first number in a range must be positive. The last number in a range specified with a hyphen must be larger than or equal to the first one. An asterisk in the first position is equivalent to the number 1. An asterisk after a hyphen specifies infinity, the end of the record, or all records in a file. Some syntax diagrams show <i>range</i> as an alternative to <i>number</i> . Though redundant, this alerts you to a difference in semantics when a number is processed differently than a range consisting of a single column. range: 

Figure 367 (Page 4 of 5). Syntactic Variables

Name	Examples	Description
<i>snumber</i>	-17 0 734298	A signed number can be positive, zero, or negative. Negative numbers have a leading hyphen; zero and positive numbers have no sign. The smallest number supported is $-2^{**}31$ (-2147483648). Note that -0 is not a <i>snumber</i> . snumber: $\begin{array}{ c } \hline \text{number} \\ \hline \text{-number} \\ \hline \end{array}$
<i>stageSep</i>		The stage separator character. By default, this is the solid vertical bar (). A different character is declared by the option SEPARATOR. Refer to Chapter 8, “Using Pipeline Options” on page 120.
<i>stream</i>	17 mstr	A number or a stream identifier. You can always refer to a particular stream by the number (the primary stream is number 0, the secondary stream number 1, and so on). Refer to a symbolic identifier instead of the stream number if a stream identifier is declared with the label (or created with the ADDSTREAM pipeline command). stream: $\begin{array}{ c } \hline \text{number} \\ \hline \text{streamID} \\ \hline \end{array}$
<i>streamID</i>	Mstr mstr	A stream identifier is a word having up to four characters. It cannot be a number. Case is respected in stream identifiers. A stream identifier made up from letters in lower case is different from one made up of the same sequence of characters in upper case.
<i>string</i>	a name	A string is a sequence of characters with or without blanks. It can have leading and trailing blanks; a string extends to the stage separator.
<i>subscript</i>	(4)	A subscript is a positive number in parentheses that is appended to an identifier that references a member of a structure. Blanks are not allowed in a subscript. While this definition covers the usage of <i>inputRanges</i> in <i>spec</i> , it does not cover subscripts in <i>spec</i> expressions; refer to “Term” on page 739. subscript: —(—number—)—
<i>word</i>	inPut 0+4	A word is a sequence of non-blank characters. Most arguments to <i>CMS Pipelines</i> filters are blank-delimited words.
<i>xorc</i>	1 F1 40 BLANK TABulate	A character specified as itself (a <i>word</i> that is one character) or its hexadecimal representation (a <i>word</i> that is two characters). The blank is represented by the keyword BLANK, which has the synonym SPACE, or with its hex value, X'40'. The default horizontal tabulate character (X'05') is represented by the keyword TABULATE, which can be abbreviated down to TAB.

Syntax

Figure 367 (Page 5 of 5). Syntactic Variables

Name	Examples	Description
<i>xrange</i>	Y X-Z 00-7f 00.256 0-00 BLANK 40-7f blank-7f blank.3 00-blank	<p>Character ranges designate the characters in the collating sequence between two specified characters; such a range is often called a hex range because the characters can be specified as <i>xorc</i>. A hex range can be written with the first and last characters separated by a hyphen ('-'), or by the first character and a count separated by a period ('.'). No blanks are allowed between the characters and the delimiters because <i>CMS Pipelines</i> scans for a <i>word</i> before scanning the word for the hex range. Hex ranges wrap from X'FF' to X'00' when the starting character is later in the collating sequence than the ending one, or the count is larger than the number of characters from the beginning character to the end of the collating sequence.</p> <p>xrange:</p> <pre> -----xorc----- -----xorc-xorc----- -----xorc.number----- </pre>

Input Range

: An input range is specified as a column range, a word range, a field range, or a member of
 : a structure.

A single column is specified by a signed number. Negative numbers are relative to the end of the record; thus, -1 is the last column of the record. A column range is specified as two signed numbers separated by a semicolon or as a *range*. When a semicolon is used, the first number specifies the beginning column and the second number specifies the ending column. When the beginning and end of a field are relative to the opposite ends of the record, the input field is treated as a null field if the ending column is left of the beginning column.

A word range is specified by the keyword WORDS, which can be abbreviated down to W. Words are separated by one or more blanks. The default blank character is X'40'. Specify the keyword WORDSEPARATOR to specify a different word separator character. WORDSEPARATOR can be abbreviated down to WORDSEP; WS is a synonym.

A field range is specified by the keyword FIELDS, which can be abbreviated down to F. Fields are separated by tabulate characters. Two adjacent tabulate characters enclose a null field. (Note the difference from words.) The default horizontal tab character is X'05'. Specify the keyword FIELDSEPARATOR to specify a different field separator character. FIELDSEPARATOR can be abbreviated down to FIELDSEP; FS is a synonym.

: The default separator characters are in effect at the beginning of a stage's operands; once a
 : separator character is changed, the change remains in effect for subsequent input ranges.

Members of Structures

A structure contains data items or embedded structures, or both. In general, a member is designated by the keyword `MEMBER` followed by the fully qualified member name, for example:

```
member s1.s2.s3.field
```

Any of the qualifiers, except the top qualifier, must be specified with a subscript if the corresponding member is an array, for example:

```
member s1.s2(4).s3.field
```

You may specify a subscript for the final member name as well, if it is an array, for example:

```
member s1.s2.s3.field(1)
```

The entire array is selected if you omit the subscript for a member that is an array.

The member name may be fully qualified, as shown above, or part of the structure qualifier may be specified by prefixing the `MEMBER` keyword with the keyword `QUALIFY` followed by the possibly qualified identifier for the structure, for example:

```
qualify s1.s2 member s3.field
```

`QUALIFY` and the qualifier may optionally be followed by a positive number, which specifies the column where the specified structure begins; the default being the beginning of the record.

Once specified, the qualifier remains in effect until a new one is specified. Use a period or a hyphen instead of the qualifier name to disable the qualifier.

You may specify two leading periods to indicate a fully qualified member name; any active qualifier is then ignored, for example:

```
member ..s1.s2.s3.field
```

You may specify a single leading period to indicate that a qualifier must be applied, for example:

```
qualify s1.s2 member .s3.field
```

In general, the qualifier applies to all input streams, though *spec* supports associating different qualifiers with each input or output stream.

The highest level structure name is resolved first in the current pipeline set; then in the containing pipeline set, and so on. Finally thread scope structures are inspected. Contained structures are resolved by *structure ADD* when the containing structure is defined.

Both `QUALIFY` and `MEMBER` can be abbreviated down to one character.

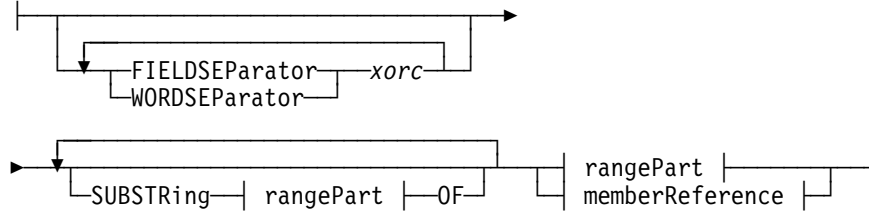
Substrings

You can select a substring of a an input range; and you can do so iteratively.

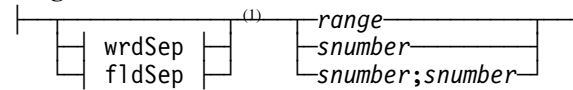
Syntax

Syntax

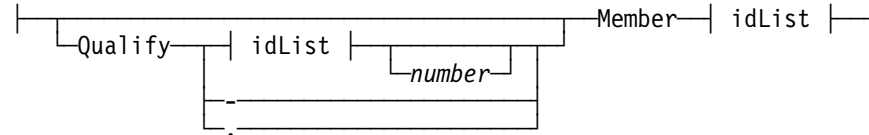
inputRange:



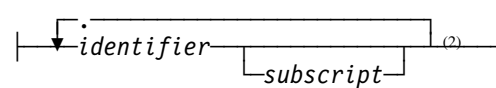
rangePart:



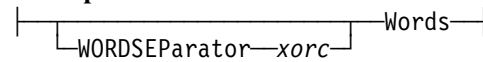
memberReference:



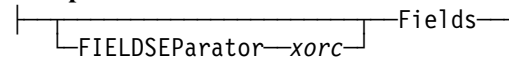
idList:



wrdSep:



fldSep:



Notes:

- ¹ Blanks are optional after the keywords WORDS and FIELDS.
- ² Blanks are not allowed in a qualified identifier or its subscript.

Examples

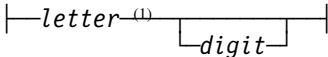
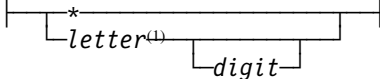
```

1-*
word 5
1;-1
-18;28
field 4
substr fs . f3 of word 7
member struct.member
member struct.member(8)
qualify struct 5 member member

```

CMS File Names

Figure 368. CMS File Names

Name	Examples	Description
<i>fn</i>	+30 some	A file name. A word with up to eight characters. CMS accepts letters, the national use characters '#@\$', numbers, plus ('+'), hyphen ('-'), colon (':'), and underscore ('_') in a file name. See “Mixed case File Names”.
<i>ft</i>	+30 some	A file type. A word with up to eight characters. CMS accepts letters, the national use characters '#@\$', numbers, plus ('+'), hyphen ('-'), colon (':'), and underscore ('_') in a file type. See “Mixed case File Names”.
<i>fmode</i>	a S2	The file mode letter followed by an optional digit. CMS accepts only the 26 characters in the English alphabet (a through z) as file mode letters. The file mode letter is translated to upper case. fmode:  Note: ¹ There is no blank between the mode letter and the mode number.
<i>fm</i>	* Q6	An asterisk, a file mode letter, or a file mode letter followed by a digit. fm:  Note: ¹ There is no blank between the mode letter and the mode number.
<i>dirid</i>	. piper.src	The directory path or a name definition for such a path. See “Shared File System Considerations” on page 232.

Mixed case File Names

The file name and file type operands for `>`, `>>`, `<`, *disk*, *diskslow*, *diskback*, *diskrandom*, *members*, *pdsdirect*, *state*, and *statew* are passed to the CMS FSSTATE macro without inspection. Likewise, the file name and file type specified for *xedit* are passed to XEDIT without inspection. The file mode is translated to upper case. The operands must designate a specific file; “wildcard” characters are not supported in the name and type except for a single asterisk ('*') in the file name or file type (or both) of *state*, *statew*, and *xedit*.

The file name and file type are translated to upper case if a file does not exist with the name as written in mixed case.

As an example, assume that these three files are stored on the minidisks or directories shown:

```
MIXED CASES A
mIXEd cAsEs B
mixed cases C
```

Syntax

Figure 369 on page 232 shows how the mode letter is resolved for particular operands.

<i>Figure 369. Mode Letters Resolved</i>	
Mode	Operands Specified
C	mixed cases
A	Mixed Cases
A	mIxEd cAses
B	mIxEd cAsEs

File Mode *

An asterisk file mode (specified or defaulted) exposes the FSSTATE search order. It searches the table of open files before accessed minidisks and directories. This can cause unexpected results as shown in Figure 370.

<i>Figure 370. File Search Order Exposed by TSTATE EXEC</i>	
/* Test State search order */ address command 'PIPE literal my own language split > system language a' 'PIPE state system language console' 'EXECIO 1 DISKR SYSTEM LANGUAGE S (SKIP' 'PIPE state system language console' 'ERASE SYSTEM LANGUAGE A'	
tstate	
▶SYSTEM	LANGUAGE A1 V 8 3 1 4/13/91 17:25:40
▶SYSTEM	LANGUAGE S2 V 5 1 1 4/01/87 14:49:39
▶Ready; T=0.01/0.01 17:25:40	

In this experiment, the file SYSTEM LANGUAGE is created on mode A by the first PIPE command. As expected, *state* resolves this file in the second command; this is the first line of output. The EXECIO command reads (and discards) one line from the file by the same name on the system disk. The third PIPE command now resolves this open file rather than the file on mode A.

Shared File System Considerations

On *CMS Pipelines* can read and write files directly in an SFS directory, which need not be accessed as a mode. *CMS Pipelines* supports the specification of a *dirid* where a mode letter is supported. The directory ID is resolved by CMS, as described in the *z/VM: CMS Commands and Utilities Reference*.

In summary, you can specify:

- A name definition. A word that contains neither period nor colon is interpreted as a name definition (see CREATE NAMEDEF). Note that a single letter or a single letter followed by a number is interpreted by *CMS Pipelines* as a mode letter and an optional number. Use name definitions that cannot be mistaken for a mode letter.
- An absolute directory path. In general, this consists of a file pool name followed by a colon, a user name followed by a period, and an optional list of directories separated by periods. You must specify the file pool if you have no default file pool established.

- A directory path relative to an accessed mode. Specify a plus (+) followed by a mode letter to start from the directory accessed as the specified mode. Specify a hyphen (-) followed by a mode letter to start from the parent directory of the directory that is accessed as the specified mode.

When the SFS interface is used, the SFS rules for sharing and updating files apply.

Though the device drivers are described as being SFS device drivers, a more correct notation would have been CSL drivers, because they use callable services, such as DMSVALDT, DMSOPEN, and DMSOPDBK; all of which support a mode letter as well as a directory path. This is not advertised (other than here) because there are subtle differences in the way CMS treats files that are open through DMSOPEN and similar, and the way CMS treats files that are accessed through the FSxxxx macro interface (which is used by the minidisk device drivers).

When a directory is accessed as a mode, it makes no difference to the SFS device drivers (for example, <*sfs*>), whether you use the mode or the directory as the third word, but it does make a difference to the router device driver (which would be <).

MVS File Names

TSO Pipelines reads and writes sequential files, and processes partitioned data sets (PDS).

A physical sequential data set is read by < and written by > or >>. A sequential data set can be either

- a physical sequential data set or
- a member of a partitioned data set (cannot be appended with >>).

Partitioned data sets are supported by:

- *readpds*, which reads specified members from a partitioned data set.
- *listpds*, which reads the directory of a partitioned data set into the pipeline, one record per member.
- *listispf*, which reads the directory of a partitioned data set into the pipeline, one record per member. If user data in the ISPF format are present in the directory record, this information is expanded to humanly readable form.
- *writepds*, which replaces members of a partitioned data set. Each member is prefixed by a delimiter record in the input stream.

TSO Pipelines supports both generation data groups and member names in DSNAME specifications. The generation is specified by a signed number in parentheses or by a zero in parentheses. When both are specified, the generation is specified before a member name:

gdg.po(+1)(member)

Syntax

Figure 371 (Page 1 of 2). MVS File Names

Name	Examples	Description
<i>ddname</i>	sysexec syspsprt	The data definition name (DDNAME) contains a letter optionally followed by one to seven characters or digits. It identifies an allocated data set.
<i>dsname</i>	'sys1.maclib' names.text 'dpjohn.tso.load'	The data set name consists of an unqualified name, which can be prefixed with qualifiers. The qualifiers are joined with a period. The maximum length of a data set name is 44 characters. A data set name is translated to upper case; <i>TSO Pipelines</i> does not support mixed case data set names.
<i>generation</i>	0 +1 -4	The particular generation of a generation data group is a signed number or zero.
<i>member</i>	FPLIRUN dump M05@45	A member name contains one to eight characters. The member name is translated to upper case. The name should not contain eight bytes of X'FF', because this indicates the end of the directory. z/OS data management does not enforce other restrictions on member names, but it is good practice to make the first character alphabetic or national use (a through z and “#@\$”) and use alpha- numerics and nationals for the following characters.

Figure 371 (Page 2 of 2). MVS File Names

Name	Examples	Description
<p><i>psds</i></p>	<p>names.text gdg(-1) pogdg(-1)(m2) dd=rexx(sample) 'sys1.maclib(time)' c admsymb1</p>	<p>A sequential data set is specified either as a physical sequential data set (DSORG=PS) or as a member of a partitioned data set (DSORG=PO). The data set can be specified by DSNAME or by DDNAME. Prefix the keyword DDNAME= to indicate that the DDNAME is specified.</p> <p>When specifying a DSNAME, the prefix is applied (if set) unless the operand is enclosed in single quotes. The trailing quote is optional.</p> <p>A relative member of a generation data group is specified by parentheses containing a signed number or zero.</p> <p>When the data set is partitioned, a member must be specified. This can be done by appending parentheses containing the member name to the data set name (after the parentheses that specify the relative generation) or by specifying two words. When two words are specified, the first word is the member name; the second word is the DDNAME. The keyword DDNAME= is an optional prefix for the second word. (This latter format is compatible with a view of the CMS file name space where a member name corresponds to a file name and a DDNAME corresponds to a file type.)</p> <p>The data set specification is translated to upper case.</p> <p>psds:</p> <pre> ----- dsname----- dsname (generation)----- dsname (member)----- dsname (generation) (member)----- ' dsname '----- ' dsname (generation) '----- ' dsname (member) '----- ' dsname (generation) (member) '----- DDname=ddname----- DDname=ddname (member)----- member ddname----- </pre>
<p><i>pods</i></p>	<p>dd=rexx 'sys1.maclib' tso.load</p>	<p>A partitioned data set as a whole can be specified by DSNAME or by DDNAME. Prefix the keyword DDNAME= to indicate that the DDNAME is specified.</p> <p>When specifying a DSNAME, the prefix is applied (if set) unless the DSNAME is enclosed in single quotes. The trailing quote is optional.</p> <p>A relative member of a generation data group is specified by parentheses containing a signed number or zero.</p> <p>The data set specification is translated to upper case.</p> <p>pods:</p> <pre> ----- dsname----- dsname (generation)----- ' dsname '----- ' dsname (generation) '----- DDname=word----- </pre>

OpenExtensions File Names

The OpenExtensions file system starts at the root directory, which is identified by a single forward slash (/). File names can contain any character except for X'00' and the forward slash. In particular, they can contain blanks and quotes.

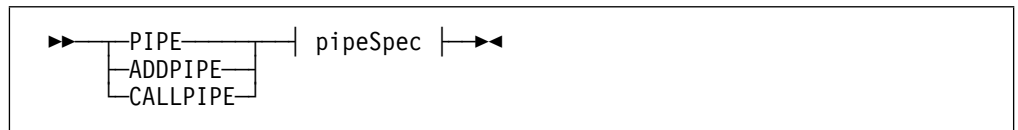
Enclose a path that contains a blank in quotes. If the path also contains one of the quotes in which you are enclosing the path, the inner quotes must be doubled. This example shows two ways to read a particular file on CMS:

```
pipe < "/the green man's directory/file one" | ...  
pipe < '/the green man''s directory/file one' | ...
```

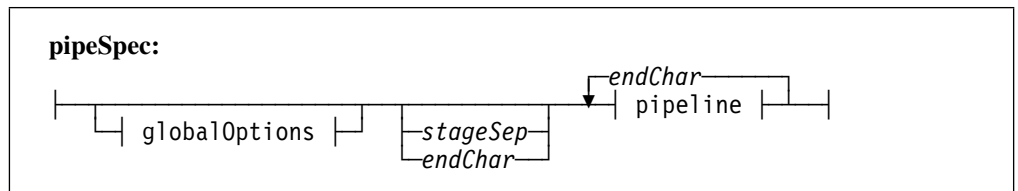
On z/OS, the use of single quotes in the second line means that the operand is to be interpreted as a reference to a fully qualified data set name; albeit not to a valid one.

Chapter 21. Syntax of a Pipeline Specification Used with PIPE, runpipe, ADDPIPE, and CALLPIPE

A pipeline specification is the argument string to the PIPE command and the format of the input records read by the *runpipe* built-in program. The pipeline commands ADDPIPE and CALLPIPE are issued from programs in a pipeline; they require a pipeline specification as the argument string.

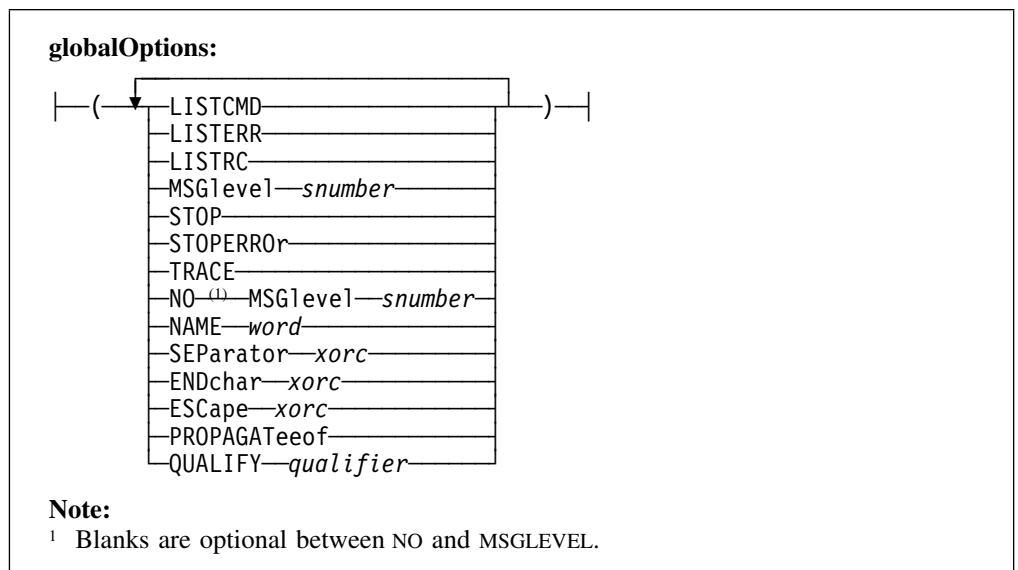


The pipeline specification is one pipeline unless an end character is declared and used to separate pipelines. You may declare an end character without using it.



In its simplest form, a pipeline specification has stages (invocations of programs) separated by stage separator characters (solid vertical bars).

Options



Pipeline options that apply to the complete pipeline specification are often referred to as *global options* in contrast to *local options*, which apply to a single stage. Pipeline options that are specified at the beginning of a pipeline specification modify the way the pipeline specification is parsed; they specify options that apply to all stages of the pipeline specification. Write global options in parentheses immediately after the command verb.

Pipeline Specification

The following options are valid only as global options. They may be specified with all three command verbs.

NAME <i>word</i>	The word is stored as a name to be used in messages. Names need not be unique. The file name of the EXEC or REXX program is recommended so that you can see the name of the program with a broken pipe when <i>CMS Pipelines</i> issues error messages.
SEPARATOR <i>xorc</i>	The character or hex value is the stage separator character for the pipeline specification. The default stage separator is the solid vertical bar. STAGESEP is a synonym for SEPARATOR.
ENDCHAR <i>xorc</i>	The character or hex value is the end character for the pipeline specification. There is no default end character.
ESCAPE <i>xorc</i>	The character or hex value is the escape character for the pipeline specification. There is no default escape character. The escape character takes effect after the parenthesis closing the global options. Local options are processed without processing for the escape character. When an escape character is used in a pipeline specification, it is deleted; the following character is not inspected for any special meaning to the scanner.

The following global option is valid with CALLPIPE only.

PROPAGATE	End-of-file propagates out through connectors, as they do for ADDPIPE. The streams are not restored after the subroutine ends; thus the subroutine should process the entire file.
-----------	--

When specified as global options, the following keywords apply to all stages of the pipeline specification. Once enabled with a global option, a keyword is disabled at a particular stage with the NO prefix.

LISTCMD	Trace pipeline commands except BEGOUTPUT, ISSUMSG, NOCOMMIT, OUTPUT, PEEKTO, READTO, and REXX.
LISTERR	Trace when a stage returns with a nonzero return code.
LISTRC	Trace when a stage begins and ends.
MSGLEVEL <i>number</i>	Specify additional bits for the message level. Bits are removed from the message level when NO is prefixed to the keyword. The message level is a fullword of switches for the pipeline dispatcher to enable checks and determine if additional messages should be issued. When specified as a global option, the new message level takes effect after the right parenthesis is scanned to close the list of global options; errors in the global options are reported as determined by the message level in effect when the command is issued.
QUALIFY <i>qualifier</i>	Specify the default qualifier for the pipeline specification. The default qualifier is inherited by encoded pipeline specifications for CALLPIPE, but not by pipeline specifications issued by other means, such as the pipeline command CALLPIPE.
STOP	Trace when a stage is started. On CMS, the virtual machine is put in CP console function mode when the pipeline dispatcher calls the syntax exit and when it calls the main entry point. Be sure to have RUN OFF.

:	STOPERROR	<p>Terminate the pipeline specification when any stage for which the option applies returns with a nonzero return code. When a stage with STOPERROR on terminates with a nonzero return code, all running stages in the pipeline specification receive return code -4094.</p> <p>You can specify NOSTOPERROR for those stages you expect to terminate with a nonzero return code, such as <i>aggrc</i>. You can also specify STOPERROR on selected stages and not use the global option (which just sets the default for all stages in the pipeline specification).</p> <p>Note that the STOPERROR option specifies which stages cause the pipeline specification to be terminated, not which stages are terminated as a result.</p>
	TRACE	<p>Trace calls to the pipeline dispatcher and trace how control passes between stages. Because this option is likely to generate large amounts of data, it is recommended that a pipeline being traced be issued with the <i>runpipe</i> built-in program.</p>

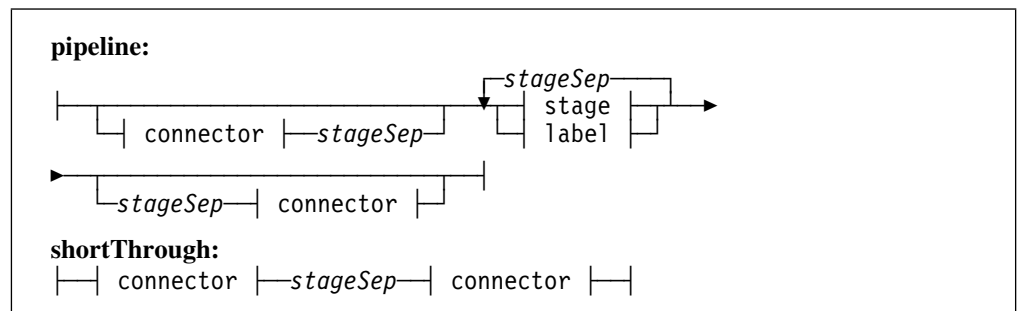
Some Bits in the Message Level

Though you can specify any number, bits other than X'000017FF' are masked off when the message level is specified as an option. The default message level is 15, corresponding to the rightmost four bits. You are likely to use only these bits:

- 4 Issue message 3 or 4 in conjunction with other messages.
- 2 Issue message 2 in conjunction with other messages from pipeline commands.
- 1 Issue message 1 in conjunction with other messages.

See “The Message Level” on page 864 for a complete description of the bits that make up the message level.

Pipeline



A pipeline contains stages and label references separated by stage separator characters. Connectors are optional at one or both sides of a pipeline issued with ADDPIPE or CALLPIPE. Connectors are delimited with stage separators.

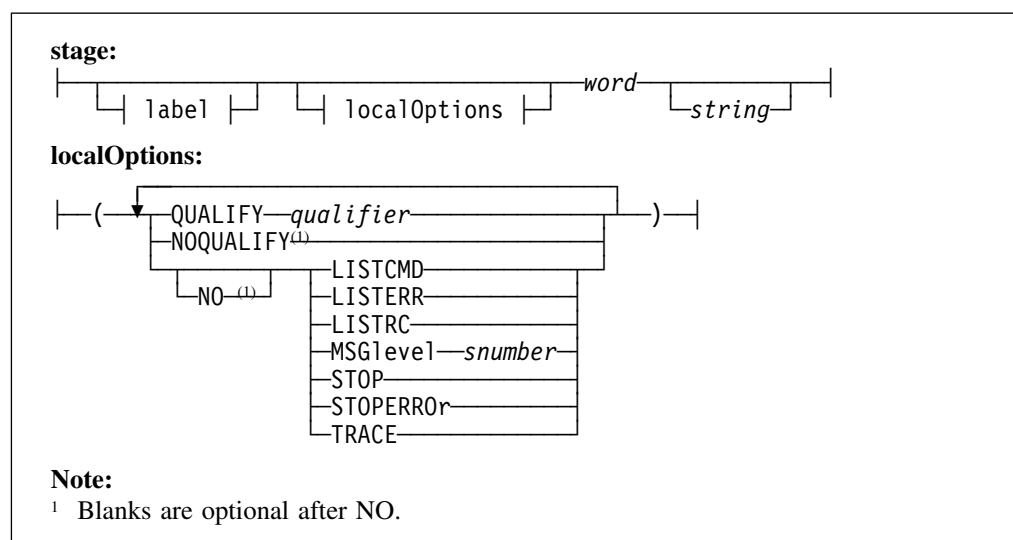
A *short-through connection* is a pipeline with two connectors and no stages. Other pipeline configurations must have at least one stage or a label reference.

Pipeline Specification

A pipeline is scanned for stage separator characters. There are several interpretations for the string between stage separators (and from the beginning or end to the nearest stage separator):

- It is a connector if it is first or last in the pipeline, begins with an asterisk, and ends with a colon.
- It is a label reference if it is one word that ends with a colon and does not start with an asterisk.
- It is the specification of a stage when it is not one of the two above.

Stage



A label is declared for a stage when the first word has a colon before the first blank or parenthesis. A label declaration beginning with a period defines the stream identifier for the primary streams; you cannot use a label reference to refer to such a label placeholder later in the pipeline. You can request a program that has a colon in its name in two ways. Define an escape character with the option ESCAPE and use this character in front of the colon, or write a dummy label, for instance, |.:am:pm|. The first colon marks the end of the label placeholder; the period separates a null word from a null stream identifier.

Write local options in parentheses after the label, if one is present. Refer to “Options” on page 237 for a description of the keywords you can specify that are also global options. In addition, option NOQUALIFY may be specified to disable the default qualifier for the stage. The options apply to the stage being defined.

The first word (after the label and local options, if any are present) is the name of the program to call. The string beginning one blank after the program name is passed to the program as the argument string. An argument string is optional; it extends to the next stage separator; it can have leading or trailing blanks, or both.

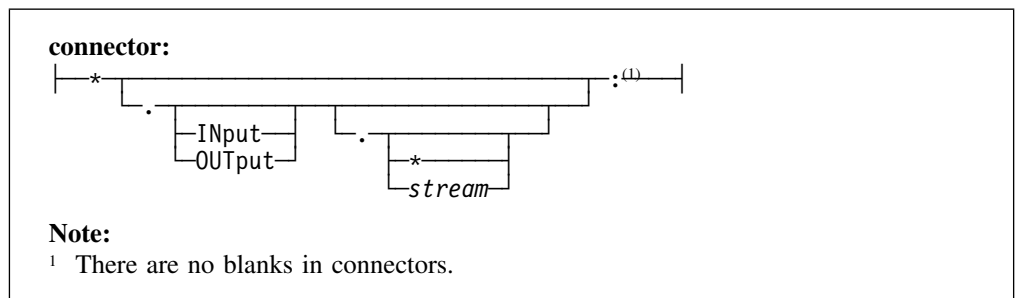
To find the entry point for the program to run, the scanner searches several entry point tables (see Appendix E, “Generating and Using Filter Packages with *CMS Pipelines*” on page 924):

- The entry point table in the PIPPTFF filter package, if it is available (that is, the filter package has been attached to the pipeline module).
- The entry point table in the main pipeline module.

- Entry point tables in filter packages that have attached their entry point tables.
- The entry point table for programs that your installation has added to the main pipeline module.

The scanner looks for a REXX program with the file name specified if the program is not resolved from any of the entry point tables. On CMS, it looks for the file type REXX or a program loaded with EXECLOAD and assigned type REXX; the *rexx* program is called to run the program if one exists. On z/OS, it searches the partitioned data set allocated to the DDNAME FPLREXX, if any.

Connectors



You can put *connectors* at the beginning or the end of a pipeline (or both) when the command is issued with ADDPIPE or CALLPIPE. Connectors refer to streams in the stage that issues the pipeline command; they specify where the streams of the stage are connected to stages in the new pipeline specification. PIPE and *runpipe* do not accept connectors because they start a new set of pipelines; there is nothing to connect to.

Syntactically, the connector is a word that begins with an asterisk ('*') and ends with a colon (':'). Two components with a leading period ('.') are optional to define the type of connector. The first component is a keyword (INPUT or OUTPUT) to specify the side of the stage; the default is INPUT at the beginning of a pipeline and OUTPUT at the end. The second component specifies the stream. It can be a number, a stream identifier, or an asterisk. An asterisk means the currently selected stream. The default is the stream currently selected.

There must be a stage separator character between the connector and the rest of the pipeline.

There are two types of connectors, *redefine* and *prefix*. They can be applied to the input and output side of a pipeline, giving four combinations.

The second component of the connector names the side it is on in a *redefine connector*. Though valid in a ADDPIPE pipeline command, a *redefine connector* is usually used in CALLPIPE pipeline commands.

`*.input:` or `*:` at the beginning of a pipeline specifies that the currently selected input stream is to be connected to the stage at the right of the stage separator ending the connector. Likewise, `*.output:` or `*:` at the end of a pipeline specifies that the currently selected output stream is to be connected to the stage at the left of the stage separator before the connector.

The new pipeline is connected to the stage issuing the ADDPIPE pipeline command in a *prefix connector*. The current connection is saved on a stack from where it is restored with

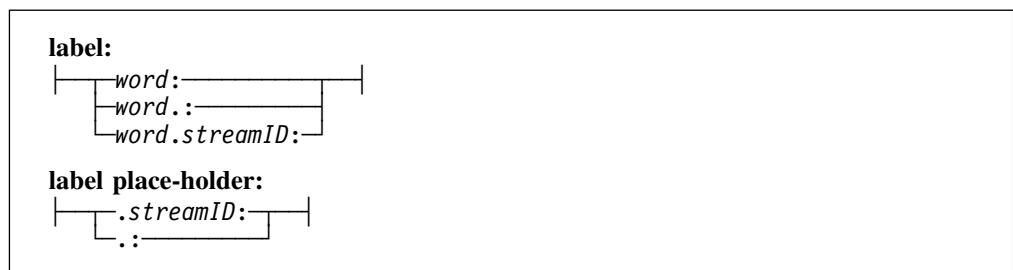
Pipeline Specification

the SEVER pipeline command. `*.input:` at the end of a pipeline specifies that the output from the new pipeline is to be connected to the currently selected input stream. Likewise, `*.output:` at the beginning of a pipeline specifies that the currently selected output stream is to be connected to the new pipeline.

A *short-through connection* has no stages between the connectors. The first one must refer to the input side; the second one must refer to the output side.

A pipeline is inserted in front of (or after) the currently selected input (output) stream when it has input (output) connectors at both ends.

Labels



A *label* is a character string, at the beginning of a stage. Case is respected in a label. It ends with a colon (':'). A label is *declared* the first time a particular label is used in a pipeline specification. The scope of a label is the pipeline specification being scanned.

Write local options followed by the name of a program to run and its argument string after the label where it is declared. This defines the primary streams for the stage. The secondary and subsequent streams for a stage with a label are defined when you reference the label later in the pipeline specification. In a *label reference*, write the label without options, program name, or arguments; they have already been specified.

A label declaration or reference may specify a stream identifier. Write a period followed by up to four characters between the label name and the ending colon. Case is respected in stream identifiers. The scope of a stream identifier is the particular stage that the label refers to. The period ending the label is optional when the stream identifier is not specified.

Example

Figure 372. Using Labels	
Pipeline Specification	Pipeline Topology
<pre> /* Sample multistream */ 'PIPE (end ? name PIPPSPEC)', ' cp query rdr * all', ' drop 1', ' search:lookup 10.4 master', ' join:faninany', ' > rdr cache a', '?disk rdr cache a', ' search:', /* Process new files */ ' > add cache a', ' join:', '?search:', ' > del cache a' </pre>	

Figure 372 shows a pipeline specification with three pipelines. It has two stages with labels, search and join. The primary output stream of *drop* is connected to the primary input stream of *lookup*. The primary output stream of *lookup* is connected to the primary input stream of *faninany*.

disk starts a new pipeline because it has an end character in front of it. Its primary output stream is connected to the secondary input stream of *lookup*. The secondary output stream of *lookup* goes through the primary stream of the second > into the secondary input stream of *faninany*. The tertiary output stream from *lookup* goes to the primary input stream of the third > stage. *lookup* has a tertiary input stream that is not connected.

Considerations when Issuing the PIPE Command

REXX Limit of 500 Characters in Clause

The REXX/MVS interpreter in TSO/E supports no more than 500 characters in a clause. This does not limit a pipeline specification to 500 characters; only virtual storage limits the complexity of a pipeline specification.

To circumvent the REXX limitation, assign parts of the pipeline to variables when a pipeline specification is longer than 500 characters. Issue the PIPE command in an expression that references these variables:

Pipeline Specification

Figure 373. Breaking a large Pipeline Specification into smaller Pieces

```
/* Huge pipeline (mostly not shown) */
part1=,
  '< input file',
  '|xlate upper'
part2=,
  '|find ABC',
  '|> output file a'

'PIPE' part1 part2
exit RC
```

Pipelines in XEDIT Macros

Always issue the PIPE by Address Command in XEDIT macros. Doing so avoids these potential pitfalls:

- XEDIT truncates commands in macros after 255 characters without diagnostic or other indication that an error has occurred. This is likely to lead to strange diagnostics when the pipeline specification is truncated.
- When PIPE is issued to XEDIT, XEDIT will look for the file PIPE XEDIT; if the file exists, it will be invoked as an XEDIT macro.

Figure 374. Issuing a PIPE Command from an XEDIT Macro

```
/* A macro */
'extract /line'
'top'
Address COMMAND 'PIPE (name PIPPSPEC)',
  '|Xedit',           /* Read current file      */
  '|split',          /* Split into words      */
  '|sort unique',    /* Find unique "words"   */
  '|count lines',    /* Count'm              */
  '|spec 1-* 1 /unique words./ nw', /* Message              */
  '|xmsg'            /* Issue it              */
':line.1
```

Chapter 22. Scanning a Pipeline Specification and Running Pipeline Programs

A pipeline is performed in two phases:

1. The pipeline specification parser, which is informally called the scanner, processes the argument string to build a control block structure describing the programs to run and their connections.
2. The pipeline dispatcher transfers control between programs to make data flow through the pipelines. The dispatcher runs only programs that are ready to run at the current commit level.

Pipeline Scanner

The pipeline specification parser first scans global options, if any are present. It stops as soon as it finds an error in the global options; the rest of the pipeline specification is not processed. When the global options have been scanned without error, the scanner performs three passes over the rest of the pipeline specification. It performs each pass to the end, reporting all errors it finds. The scanner terminates at the end of a pass if it finds errors.

1. Determine the overall structure of the pipeline specification. The scanner counts connectors, stages, and pipelines. Errors detected include null stages and null pipelines. If no errors were found on the first pass, the scanner then allocates storage for a control block to represent the pipeline specification and all its stages, streams, and connectors.
2. Resolve labels and entry points. At this pass, the control blocks are filled with information from the argument string. Errors detected include unresolved entry points, undefined labels, and labels that are defined more than once.
3. Check the placement and argument syntax for entry points that are resolved to a program descriptor (the expansion of the PIPDESC macro). This applies to all built-in programs. If it is requested in the program descriptor, the scanner calls the stage's syntax exit to process the argument string. The entire pipeline specification is suppressed if the scanner detects an error in the syntax of any one stage or if any syntax exit returns a return code that is not zero.

When the scanner has completed the third pass without finding errors, it hands the pipeline specification over to the pipeline dispatcher to perform the work to be done.

Pipeline Dispatcher

The main function of the dispatcher is to run a stage and regain control when the stage requests a pipeline service or terminates.

Each stage runs independently of other stages, because a stage calls the dispatcher to read or write, rather than calling the neighbour stages to obtain or deliver records. This division of labour has many advantages, the most obvious one being that all stages use a standard interface to the dispatcher. A more subtle advantage is that each stage's call stack is usually quite shallow: the stage often calls a few internal subroutines and the dispatcher, but it is not entered recursively.

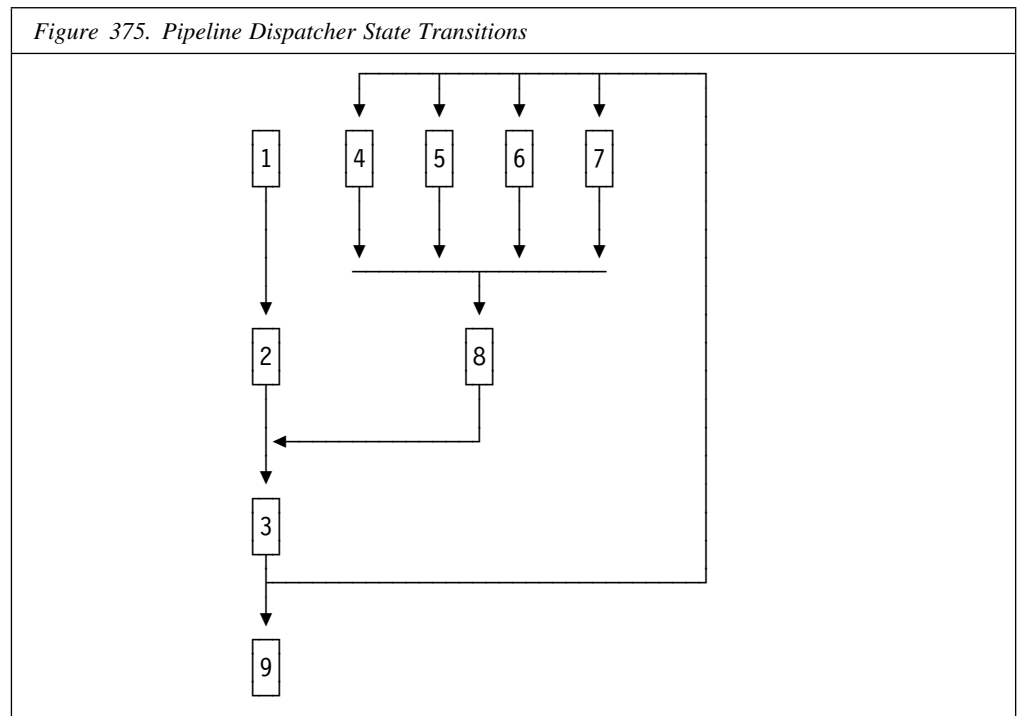
States

States of a Stage

A stage goes through these states during its lifetime (see Figure 375 on page 247 for a diagram):

1. Committed to start. When the scanner hands the pipeline specification over to the dispatcher, all stages are waiting to start. The dispatcher commits to the lowest level where a stage is committed to start and makes the stages on this commit level ready to run. (See “Commit Level” on page 247.)
2. Ready, not started. The first time the stage is run, it is *started*. The stage’s environment is set up and the main entry point is called.
3. Running. One stage is running at a time. Once the stage is given control, it runs until it gives up control voluntarily. It can give up control in one of the following ways: It can issue a service request to the dispatcher to transport data, change the pipeline topology, or wait for an external event. If the stage cannot continue immediately (or the dispatcher decides that it should not), the stage is made not dispatchable until the condition that blocks its progress has been cleared.
4. Waiting for pipeline I/O. A stage waits for I/O, for example, when it performs a read operation and there is no data available to read on the currently selected input stream.
5. Waiting for a subroutine pipeline to complete. A stage that has issued a CALLPIPE pipeline command waits until all stages in the subroutine pipeline have ended and all connections are restored.
6. Waiting to commit. A stage that issues the COMMIT pipeline command to commit to a level that is higher than the current commit level must wait. The stage is made dispatchable when the pipeline specification has committed to this level.
7. Waiting for an external event. The stage issues the PIPWECB macro. For example, *delay* waits for a timer interrupt. This interface is not available to REXX filters; no pipeline command can cause a stage to wait for an external event.
8. Ready. The stage can run, but it is not currently running. A ready stage is said to be on the *run list*.
9. Terminated. That is, the stage has returned on the original call from the dispatcher. This means that the stage has completed the task it was set to perform, that the stage has determined that it can perform no more useful work, or that the stage has failed. In the latter case, the stage sets a nonzero return code.

The dispatcher will at some future time *resume* a stage that is waiting (a stage that is in one of the states 4 through 7). That is, the dispatcher will return to the stage to the next instruction after the call to the dispatcher service. The stage sees a dispatcher service as synchronous; no activity takes place in the stage during the call to the dispatcher service.



Commit Level

The *commit level* provides a general mechanism to allow unrelated programs to coordinate their progress. One use of the commit level mechanism is to allow all stages to validate their argument strings before any stage takes an action that might potentially destroy data, such as erasing a file or writing on a tape. Thus, the pipeline is abandoned if a built-in program detects an error in its arguments or if a REXX program returns with a nonzero return code before reading or writing.

The commit level is a number between -2147483647 and +2147483647 inclusive. Each stage is at a particular commit level at any time. It increases its commit level with the pipeline command COMMIT. It cannot decrease its commit level. The pipeline specification parser performs an implicit commit when a stage is defined. The program descriptor for a built-in program includes the commit level at which the program begins; selection stages begin at commit level -2; REXX stages begin at commit level -1; by default, other stages begin at commit level 0.

The pipeline dispatcher initiates the stage with the lowest commit level first. When more than one stage begins at a particular commit level, it is unspecified which one runs first. The stages at the lowest commit level run until they complete (exit or return) or issue a COMMIT pipeline command.

An *aggregate return code* is associated with a pipeline specification. Initially, the aggregate return code is zero. The aggregate return code for the pipeline specification is updated with the return code as each stage returns. If either number is negative, the aggregate return code is the minimum of the two numbers; otherwise, it is the maximum.

When all stages at the lowest commit level have ended or committed to a higher level, the stages at the next commit level are examined. Stages that would begin at the new commit level are abandoned if the aggregate return code is not zero. For stages that are waiting to commit to the new commit level, the return code for the COMMIT pipeline command is set to the aggregate return code; those stages are then made ready to run. The aggregate

Commit Level

return code is sampled at the time the pipeline specification is raised to the new commit level. All stages committing to a particular level see the same return code, even if one of them subsequently returns with a nonzero return code before another stage has begun to run at the new level. A stage can inspect the COMMIT return code and perform whatever action is required; built-in programs deallocate resources they have allocated and return with return code zero when the COMMIT return code is not zero, thus quitting when they determine that another stage has failed.

By convention, all built-in programs process data on commit level 0. Stages must be at the same commit level for data to pass between them, except when data flow on a connection that has been set up with ADDPIPE. The pipeline stalls if a stage at one commit level reads or writes a record after the stage at the other side of the connection has issued a COMMIT pipeline command to commit to a higher level.

The scope of the commit level is a pipeline specification. Pipelines added with ADDPIPE commit without coordinating their commit level with the pipeline that added them. Pipeline specifications that are issued with CALLPIPE and contain no connectors (an *unconnected pipeline specification*) also commit without coordination with the caller.

When a pipeline specification that is issued with CALLPIPE (and is connected to its caller) increases its commit level, the pipeline dispatcher checks that the commit level for the stage that issued the CALLPIPE is at or above the new level requested. When the subroutine would go to a commit level that is higher than the caller's current commit level, the pipeline dispatcher performs an implicit commit for the stage that issued the CALLPIPE. The subroutine pipeline proceeds only after the caller's commit has completed (that is, only after the commit level of the calling pipeline has been raised to the new level). If the caller is itself in a subroutine pipeline, the new commit level propagates upwards.

A REXX pipeline stage begins at commit level -1. The commit level for a REXX stage is automatically raised to level 0 when it first issues an OUTPUT, PEEKTO, READTO, or SELECT ANYINPUT pipeline command. Because the pipeline dispatcher raises the commit level automatically, most REXX programs need not be concerned with commit levels. In the usual case, a REXX program validates its arguments before it begins reading and writing data. If it finds an error in its arguments and exits with an error return code before it has used any of the four commands that cause an automatic commit, the pipeline specification will in effect terminate at commit level -1, before data have begun flowing and before other stages have taken any irreversible actions (assuming they adhere to the convention of doing such on commit level 0). On the other hand, if a REXX program finds no error in its arguments and begins to process data by using one of these four commands, the automatic commit is done, suspending that stage until all other stages are ready for data to flow.

In some cases the automatic setting of the commit level for REXX programs may not be suitable. If your REXX program erases files or performs some other irreversible function before it reads or writes, it should first use the COMMIT pipeline command to do an explicit commit to level 0 to wait until all other stages have validated their arguments. If the return code on COMMIT is not zero, the program should undo any changes it may have made and exit with return code 0.

If your REXX program needs to use any of the commands that cause an automatic commit before it is ready to commit to level 0, it must issue the NOCOMMIT pipeline command to disable the automatic commit and then later issue an explicit COMMIT. To perform read or write operations on commit level -1 (to read a parameter file, for example), use ADDPIPE to connect the input or output stream (or both) to your REXX stage. (You cannot use CALLPIPE for this, because it would force a commit to level 0 before data could flow.)

Having defined the new streams with ADDPIPE, use READTO and OUTPUT to read and write. When you are finished, issue SEVER to restore the original connection. Then issue COMMIT to perform an explicit commit. Check the return code on the COMMIT before reading or writing the original stream.

The pipeline dispatcher runs stages if all syntax checks complete without reporting any errors. The order of dispatching at any commit level is unspecified. The pipeline dispatcher does not preempt stages; once a stage is running, the pipeline dispatcher regains control in one of two ways:

- The program calls a pipeline dispatcher entry point, for instance to read a record.
- The program completes and returns from the initial call.

Reading, Writing

CMS Pipelines transports records without buffering from an output stream of one stage to an input stream of another stage.

To write a record, a program (the *producer stage*) calls the pipeline dispatcher with the address and length of a buffer that contains the record to be written; the equivalent pipeline command is OUTPUT. The stage is then blocked (cannot run) until the neighbour to the right (the *consumer stage*) performs an action that releases the producer:

- It reads (*consumes*) the record by calling the pipeline dispatcher. This sets return code 0 on the producer's write and makes the producer able to run.
- It severs the input stream that is connected to the producer's output stream. The dispatcher, in turn, sets return code 12 on the producer's write to indicate end-of-file and makes the producer able to run.
- It returns on the initial invocation from the dispatcher, because processing is complete or abandoned. The dispatcher then severs all the terminating stage's streams and sets end-of-file on all reads and writes that are waiting for the terminating stage to produce or consume a record.

A stage waits for a record to become available if there is none at the time it reads. There are two ways to read records. The simplest is to call the pipeline dispatcher, passing the address and length of a buffer where the next record is placed; this is done by READTO in a REXX filter. If the neighbour on the left is blocked waiting for a record to be read, the record is copied and both stages are made ready to run. This type of read is called a *consuming read*, because the read has consumed the record. It is also called a *move mode read*, because the record is moved into the reading stage's buffer.

Move mode reads are not well suited to programs that must process records of any size. Instead, such a stage first performs a *locate mode read* to determine the length of the record; the address and length of the producer's buffer are returned: the producer remains blocked waiting for the record to be consumed. (Move mode and locate mode are terms from OS data management).

The pipeline command PEEKTO in a REXX filter performs a locate mode read. The program then issues the pipeline command READTO when it has processed the record; this releases the neighbour on the left. READTO is normally issued without specifying a variable name, which corresponds to a move mode read with buffer length zero. Unlike OS data management, the same record is returned on multiple locate mode read calls with no intervening consuming read to release the record.

Record Delay

By using a locate mode read, a stage can peek at the first record of a file and choose a suitable subroutine pipeline to process the file, for example, to unpack the file if it is packed. The subroutine pipeline also sees the record that determined the strategy, because the first record is not consumed by the peeking stage and is thus available to the subroutine.

Delaying the Record

When you are writing an application that uses multistream pipelines, it is often important that you be able to reason about the way records move through the pipeline network relative to each other. You may wish to:

- Be sure that a record taking one path through the network cannot be overtaken by a record that takes some other path. (Otherwise the output file might be out of sequence.)
- Be sure that all records at the input streams of *specs* are available concurrently. (Otherwise the pipeline network might stall.)

This section introduces the concept of record delay, which is not a temporal delay; it explains how you can reason about record delays in cascades of stages and in other topologies.

As you have seen several times, the order of dispatching (the sequence in which the dispatcher runs stages) is unspecified. To make the order predictable, you must ensure that the dispatcher has no choice: if it has only one stage it can run, the dispatcher must run this stage however unpredictable it tries to be.

The term *record delay* specifies the degree of control that a program can exert over the pipeline dispatcher.

A program that does not delay the record processes the file in this way:

1. It obtains an input record with a locate mode read. The PEEKTO pipeline command is used in a REXX program for this purpose. This blocks the stage that produced the record.
2. It processes the record. For example, a device driver copies the record to the host interface or into a buffer; a filter, perhaps, selects a substring of the record or it copies the record into a buffer to be modified; and a selection stage determines which output stream to use.
3. It writes one output record. The record can be in a buffer that the stage has obtained, or it can be in the buffer provided by the producer stage (if the contents of the record have not been modified).
4. It consumes the input record. The READTO pipeline command is used in a REXX program for this purpose. The producer stage can resume and run in parallel with the consuming stage, but not for long; as soon as the consuming stage performs a locate mode read, it will be blocked until the producer writes the next record.

Because the producer stage is blocked while the record is written in step 3, a program that processes a record in this way does not allow the producer to produce one more record until the consumer's output record has been consumed. You can prove by induction that a cascade of stages that do not delay the record behaves in the same way as a single stage that does not delay the record. You can also prove that, for each input record, a decoding network (see "Decoding Trees" on page 82) composed entirely of stages that do not delay

the record produces a record on one stream, and on one stream only, when the secondary output streams are connected in all selection stages.

When records take different paths from a common stage (for example *fanout* or a selection stage) through a multistream network consisting entirely of stages that do not delay the record, the records will arrive at the end of this network in the same order as they entered. This is clearly a desirable property.

A program that consumes the input record before producing the output record (steps 3 on page 250 and 4 on page 250 are performed in reverse order) has the *potential to delay a record*, because it allows the dispatcher to resume the producer stage. Whether the record is, in fact, delayed will depend on the dispatching strategy, which is unspecified. When a producer stage produces records on several streams that eventually are connected to the inputs of a stage that *synchronises* its input streams (that is, the program performs a locate mode read on all its input streams before processing the records), a record delay is *required* on all but the highest-numbered stream to avoid a stall. The dispatcher will eventually run the producer stage to produce one more record before the consumer's record is consumed (by its consumer, in turn).

A program that reads a record into a buffer, consumes it, and then performs a locate mode read before it produces an output record unconditionally delays one record. But when such a program is used on a subset of a file (because other records take a different path that shunts the delaying stage), a delay of one record in the program will, in general, lead to an indeterminable delay in the file as a whole.

The strict definition above of a stage that does not delay the record stipulates that a program must produce exactly one output record for each input record.

Though the strict definition is required when one reasons about multistream networks where the contents of a record are written to more than one output stream (*chop* or *fanout*) and gathered with a program that synchronises its input streams, a slightly relaxed behaviour may be sufficient to reason about topologies where records are gathered with *faninany*. In step 3 on page 250, it may be acceptable that no record is produced (thus, the stage will delete or discard an input record); or it may be acceptable that several output records are produced as long as these records are produced before the corresponding input record is consumed.

It is noted in the descriptions of the built-in programs which ones strictly do not delay the record and which programs produce all output derived from an input record before the record is consumed.

The description of a built-in program can also specify that the program has the potential to delay one record.

Device Drivers that Wait for External Events

Most device drivers use synchronous CMS interfaces to read and write host interfaces. When using such interfaces, all pipeline stages are suspended while CMS accesses the host interface.

A few stages, however, wait for external events; *CMS Pipelines* is able to run other stages while these programs wait for external events: *console* ASYNCHRONOUSLY (but not the other two ways to read from the console), *delay*, *fullscr* (on CMS and under certain conditions), *immcmd*, *starmsg*, *tcpclient*, *tcpdata*, *tcplisten*, and *udp*.

Return Codes

Return Codes

When one or more error messages are issued by the pipeline specification parser, the return code from PIPE is the “worst” of the ones found. If any return code is negative, the worst return code is the most negative return code received; otherwise the return code is the maximum of the return codes received.

When a stage terminates because of an error in arguments or data, the return code is, in general, equal to the number of the error message issued.

Return code -7 from the environment processing a pipeline command means that the argument is not recognised as a pipeline command. Refer to “Return Codes -3 and -7” on page 116.

Return code -9 on the PIPE command means that storage was not available for the work area and save area. No explanatory message is issued because of the lack of storage.

Return code -4095 is reflected to the stages by the pipeline dispatcher when the pipelines are stalled. Messages list the status of each stage.

Chapter 23. Inventory of Built-in Programs

This chapter specifies the syntax and semantics of the built-in programs in *CMS Pipelines*. Definitions are in alphabetical order with special characters first. Each definition consists of:

- A synopsis in bold type having the name of the program in the left margin.
- A short description of the main function performed by the program.
- A syntax diagram showing how to invoke the program. The notation is defined in Chapter 20, “Syntax Notation” on page 222.
- The type of program:
 - *Controls* perform a function depending on the data at hand; a control stage can also redefine the pipeline topology while running a filter or device driver.
 - *Device drivers* interface between the pipeline and the host system.
 - *Filters* perform an operation on a single stream. They do not use host interfaces.
 - *Gateways* interface to multiple streams.
 - *Look up routines* find entry points that are not resolved as part of the standard *CMS Pipelines* resolution for stages.
 - *Host command processors* send input lines to host command environments; some intercept command output and write it to the pipeline; others pass the command on when control returns from the host interface.
 - *Sorters* sort, merge, collate, or in other ways order records by comparing the contents of key fields.
 - *Selection stages* read the primary input stream and write records to the primary output stream or the secondary output stream, depending on the contents of the record, its position in the file, or some other condition. When both output streams are connected for a selection stage, an input record is written **once** to exactly one stream.
 - *Service programs* do not read the pipeline; they perform some other service. Service programs do not require an output stream; but if the output stream is connected, the response is written to the output rather than to the terminal.

A specialised program is marked *arcane*. Such a program has a specialised purpose or accesses an interface that may be obscure.

- The placement, when a program cannot be used in all positions of a pipeline. Some programs must be first, others must not be first. The majority of built-in programs do not inspect their position in the pipeline.
- A verbal description of the syntax, if the program accepts or requires arguments.
- A description of the operation of the program, if the initial description does not completely specify the program’s operation.
- The format to which input records must adhere, if input data are structured in some way.
- The format of output records produced, if they are structured.
- A summary of streams used, if the program references more than the primary input and output streams.

Built-in Programs

- The record delay, if applicable. It is specified under which conditions the program does or does not delay the record. When no such clause is present, it is unspecified whether the program delays the record; the program may delay some records but not others.
- The commit level at which the program starts (if it starts before level 0). This part also describes the actions performed before the program commits to level 0.
- The conditions under which the program will *terminate prematurely*; that is, without processing all available input records or without producing all possible output records. A program terminates normally when all its input streams are at end-of-file, or (in the case of a device driver that is first in a pipeline) the host interface signals end-of-file or a similar condition. When a program is described as *not terminating normally*, it means that the program accesses a host interface that does not signal end-of-file; if not terminated prematurely the program will run forever.
- A reference to the converse operation that reverses the effect of the program, where one exists.
- References to programs that perform a related function, if any are provided.
- Examples of usage. Examples that show a PIPE command followed by output lines marked with an arrowhead were run as this book was formatted; you may have some confidence that they run with the *CMS Pipelines* level described by this book (1.1.12/12).

```
!      pipe query level
!      ▶CMS Pipelines, 5741-A07 level 110C0011
!      ▶Ready;
```

Examples with a leading comment line are fragments of REXX programs. Other examples show a few stages of a pipeline; they are usually a single line which begins and ends with an ellipsis (...) to indicate the remaining part(s) of the pipeline.

- Notes, where applicable.
- Return codes issued where they do not represent *CMS Pipelines* messages. Most of these are return codes from CMS.
- Configuration variables that apply to the built-in program. The main description of the built-in program will assume the PIPE style; any differences in other styles are noted in this section. See also Chapter 28, “Configuring *CMS Pipelines*” on page 867.

The built-in programs are fussy about their arguments. Quietly ignoring excess parameters can be disastrous. An unexpected parameter could be the beginning of what should have been a following stage, where the stage separator is missing.

Overview by Category

The following tables list the built-in programs by task or function. New built-in programs are not marked with a change bar in this section; refer to the index for an overview of new programs.

Figure 376. Controls

	<i>append</i>	Put Output from a Device Driver after Data on the Primary Input Stream.
	<i>casei</i>	Run Selection Stage in Case Insensitive Manner.
	<i>eofback</i>	Run an Output Device Driver and Propagate End-of-file Backwards.
	<i>frtarget</i>	Select Records from the First One Selected by Argument Stage.
	<i>not</i>	Run Stage with Output Streams Inverted.
	<i>pipcmd</i>	Issue Pipeline Commands.
	<i>pipestop</i>	Terminate Stages Waiting for an External Event.
	<i>preface</i>	Put Output from a Device Driver before Data on the Primary Input Stream.
	<i>runpipe</i>	Issue Pipelines, Intercepting Messages.
	<i>totarget</i>	Select Records to the First One Selected by Argument Stage.
	<i>zone</i>	Run Selection Stage on Subset of Input Record.
	<i>filterpack</i>	Manage Filter Packages.

Figure 377 (Page 1 of 4). Device drivers

	<i>waitdev</i>	Wait for an Interrupt from a Device.
Access to variables	<i>rexvars</i>	Retrieve Variables from a REXX or CLIST Variable Pool
	<i>stem</i>	Retrieve or Set Variables in a REXX or CLIST Variable Pool.
	<i>sysvar</i>	Write System Variables to the Pipeline.
	<i>var</i>	Retrieve or Set a Variable in a REXX or CLIST Variable Pool.
	<i>vardrop</i>	Drop Variables in a REXX Variable Pool.
	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
	<i>varload</i>	Set Variables in a REXX or CLIST Variable Pool
	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
CMS files	<i><mnsk</i>	Read a CMS File from a Mode.
	<i>>>mnsk</i>	Append to or Create a CMS File on a Mode.
	<i>>mnsk</i>	Replace or Create a CMS File on a Mode.
	<i>aftfst</i>	Write Information about Open Files.
	<i>mnskback</i>	Read a CMS File from a Mode Backwards.
	<i>mnskfast</i>	Read, Create, or Append to a CMS File on a Mode.
	<i>mnskrandom</i>	Random Access a CMS File on a Mode.
	<i>mnskslow</i>	Read, Append to, or Create a CMS File on a Mode.
	<i>mnskupdate</i>	Replace Records in a File on a Mode.
	<i>state</i>	Provide Information about CMS Files.
	<i>statew</i>	Provide Information about Writable CMS Files.
CMS libraries	<i>members</i>	Extract Members from a Partitioned Data Set.
	<i>pdsdirect</i>	Write Directory Information from a CMS Simulated Partitioned Data Set.
CP	<i>acigroup</i>	Write ACI Group for Users.
	<i>trfread</i>	Read a Trace File.
CP	<i>devinfo</i>	Write Device Information.
CP	<i>diage4</i>	Submit Diagnose E4 Requests.
FBA disk	<i>fbaread</i>	Read Blocks from a Fixed Block Architecture Drive.
	<i>fbawrite</i>	Write Blocks to a Fixed Block Architecture Drive.

Built-in Programs

Figure 377 (Page 2 of 4). Device drivers

Files	< > >> <i>diskback</i> <i>diskfast</i> <i>diskrandom</i> <i>diskslow</i> <i>diskupdate</i> <i>getfiles</i>	Read a File. Replace or Create a File. Append to or Create a File. Read a File Backwards. Read, Create, or Append to a File. Random Access a File. Read, Create, or Append to a File. Replace Records in a File. Read Files.
Hardware	<i>stfle</i> <i>stsi</i>	Store Facilities List. Store System Information.
Minidisk	<i>trackread</i> <i>trackwrite</i>	Read Full Tracks from ECKD Device. Write Full Tracks to ECKD Device.
MVS libraries	<i>listispf</i>	Read Directory of a Partitioned Data Set into the Pipeline.
MVS SPOOL	<i>sysout</i>	Write System Output Data Set.
Network	<i>ftp</i> <i>tcpclient</i> <i>tcpdata</i> <i>tcplisten</i> <i>udp</i>	Connect to an FTP Server and Exchange Data. Connect to a TCP/IP Server and Exchange Data. Read from and Write to a TCP/IP Socket. Listen on a TCP Port. Read and Write an UDP Port.
Network	<i>hostid</i> <i>hostname</i>	Write TCP/IP Default IP Address. Write TCP/IP Host Name.
OpenExtensions	< <i>oe</i> >> <i>oe</i> > <i>oe</i> <i>filedescriptor</i> <i>hfs</i> <i>hfsdirectory</i> <i>hfsquery</i> <i>hfsreplace</i> <i>hfsstate</i> <i>hfsexecute</i>	Read an OpenExtensions Text File. Append to or Create an OpenExtensions Text File. Replace or Create an OpenExtensions Text File. Read or Write an OpenExtensions File that Is Already Open. Read or Append File in the Hierarchical File System. Read Contents of a Directory in a Hierarchical File System. Write Information Obtained from OpenExtensions into the Pipeline. Replace the Contents of a File in the Hierarchical File System. Obtain Information about Files in the Hierarchical File System. Issue OpenExtensions Requests.
SFS files	<i>sfsdirectory</i>	List Files in an SFS Directory.
Shared files	< <i>sfs</i> < <i>sfsslow</i> >> <i>sfs</i> >> <i>sfsslow</i> > <i>sfs</i> <i>filetoken</i> <i>sfsback</i> <i>sfsrandom</i> <i>sfsupdate</i>	Read an SFS File. Read an SFS File. Append to or Create an SFS File. Append to or Create an SFS File. Replace or Create an SFS File. Read or Write an SFS File That is Already Open. Read an SFS File Backwards. Random Access an SFS File. Replace Records in an SFS File.
Tape	<i>tape</i>	Read or Write Tapes.

Figure 377 (Page 3 of 4). Device drivers

Terminal	<i>browse</i> <i>console</i> <i>fullscr</i> <i>fullscrq</i> <i>fullscrs</i>	Display Data on a 3270 Terminal. Read or Write the Terminal in Line Mode. Full screen 3270 Write and Read to the Console or Dialed/Attached Screen. Write 3270 Device Characteristics. Format 3270 Device Characteristics.
VM SPOOL	<i>printmc</i> <i>punch</i> <i>reader</i> <i>uro</i> <i>xab</i>	Print Lines. Punch Cards. Read from a Virtual Card Reader. Write Unit Record Output. Read or Write External Attribute Buffers.
VMCF	<i>vmcdata</i> <i>vmclient</i> <i>vmclisten</i>	Receive, Reply, or Reject a Send or Send/receive Request. Send VMCF Requests. Listen for VMCF Requests.
WebSphere MQ	<i>mqsc</i>	Issue Commands to a WebSphere MQ Queue Manager.
XEDIT	<i>xedit</i> <i>xmsg</i>	Read or Write a File in the XEDIT Ring. Issue XEDIT Messages.
Z/OS data set	<i><mvs</i> <i>>>mvs</i> <i>>mvs</i> <i>listcat</i> <i>state</i> <i>sysdsn</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set Append to a Physical Sequential Data Set Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set Obtain Data Set Names. Verify that Data Set Exists. Test whether Data Set Exists.
Z/OS libraries	<i>listdsi</i> <i>readpds</i> <i>writepds</i>	Obtain Information about Data Sets. Read Members from a Partitioned Data Set Store Members into a Partitioned Data Set
Z/OS libraries CMS libraries	<i>listpds</i>	Read Directory of a Partitioned Data Set into the Pipeline.

Built-in Programs

Figure 377 (Page 4 of 4). Device drivers

Other	<i>3277enc</i>	Write the 3277 6-bit Encoding Vector.
	<i>beat</i>	Mark when Records Do not Arrive within Interval.
	<i>delay</i>	Suspend Stream.
	<i>emsg</i>	Issue Messages.
	<i>hole</i>	Destroy Data.
	<i>immcmd</i>	Write the Argument String from Immediate Commands.
	<i>ispf</i>	Access ISPF Tables.
	<i>literal</i>	Write the Argument String.
	<i>mdiskblk</i>	Read or Write Minidisk Blocks.
	<i>pause</i>	Signal a Pause Event.
	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
	<i>random</i>	Generate Pseudorandom Numbers.
	<i>sql</i>	Interface to SQL.
	<i>sqlcodes</i>	Write the last 11 SQL Codes Received.
	<i>sqlselect</i>	Query a Database and Format Result.
	<i>stack</i>	Read or Write the Program Stack.
	<i>storage</i>	Read or Write Virtual Machine Storage.
	<i>strliteral</i>	Write the Argument String.
	<i>vmc</i>	Write VMCF Reply.
	<i>xrange</i>	Write a Range of Characters.

Figure 378. Drivers

	<i>diskid</i>	Map CMS Reserved Minidisk.
--	---------------	----------------------------

Figure 379 (Page 1 of 3). Filters

	<i>3277bfra</i>	Convert a 3270 Buffer Address Between Representations.
	<i>aggrc</i>	Compute Aggregate Return Code.
	<i>combine</i>	Combine Data from a Run of Records.
	<i>count</i>	Count Lines, Blank-delimited Words, and Bytes.
	<i>dateconvert</i>	Convert Date Formats.
	<i>duplicate</i>	Copy Records.
	<i>escape</i>	Insert Escape Characters in the Record.
	<i>greg2sec</i>	Convert a Gregorian Timestamp to Second Since Epoch.
	<i>reverse</i>	Reverse Contents of Records.
	<i>scm</i>	Align REXX Comments.
	<i>sec2greg</i>	Convert Seconds Since Epoch to Gregorian Timestamp.
	<i>timestamp</i>	Prefix the Date and Time to Records.
	<i>crc</i>	Compute Cyclic Redundancy Code.
	<i>tcpcksum</i>	Compute One's complement Checksum of a Message.
Assembler files	<i>asmcont</i>	Join Multiline Assembler Statements.
	<i>asmxpnd</i>	Expand Joined Assembler Statements.
Buffering	<i>buffer</i>	Buffer Records.
	<i>copy</i>	Copy Records, Allowing for a One Record Delay.
	<i>instore</i>	Load the File into a storage Buffer.
	<i>outstore</i>	Unload a File from a storage Buffer.

<i>Figure 379 (Page 2 of 3). Filters</i>		
Buffering	<i>noeofback</i>	Pass Records and Ignore End-of-file on Output.
Compiler	<i>polish</i>	Reverse Polish Expression Parser.
Cut and paste	<i>chop</i>	Truncate the Record.
	<i>join</i>	Join Records.
	<i>joincont</i>	Join Continuation Lines.
	<i>pad</i>	Expand Short Records.
	<i>snake</i>	Build Multicolumn Page Layout.
	<i>spill</i>	Spill Long Lines at Word Boundaries.
	<i>split</i>	Split Records Relative to a Target.
	<i>strip</i>	Remove Leading or Trailing Characters.
Data conversion	<i>utf</i>	Convert between UTF-8, UTF-16, and UTF-32
Digest	<i>digest</i>	Compute a Message Digest.
Encryption	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher.
Format conversion	<i>64encode</i>	Encode to MIME Base-64 Format.
	<i>addrdw</i>	Prefix Record Descriptor Word to Records.
	<i>apldecode</i>	Process Graphic Escape Sequences.
	<i>aplenode</i>	Generate Graphic Escape Sequences.
	<i>block</i>	Block to an External Format.
	<i>buildscr</i>	Build a 3270 Data Stream.
	<i>deblock</i>	Deblock External Data Formats.
	<i>fblock</i>	Block Data, Spanning Input Records.
	<i>fntfst</i>	Format a File Status Table (FST) Entry.
	<i>iebcopy</i>	Process IEBCOPY Data Format.
	<i>ip2socka</i>	Build sockaddr_in Structure.
	<i>pack</i>	Pack Records as Done by XEDIT and COPYFILE
	<i>parcel</i>	Parcel Input Stream Into Records.
	<i>qpdecode</i>	Decode to Quoted-printable Format.
	<i>qpenode</i>	Encode to Quoted-printable Format.
<i>socka2ip</i>	Format sockaddr_in Structure.	
<i>unpack</i>	Unpack a Packed File.	
<i>urldeblock</i>	Process Universal Resource Locator.	
Format conversion	<i>64decode</i>	Decode MIME Base-64 Format.
Other	<i>hlasm</i>	Interface to High Level Assembler.
Other	<i>hlasmerr</i>	Extract Assembler Error Messages from the SYSADATA File.
Printer files	<i>asatomc</i>	Convert ASA Carriage Control to CCW Operation Codes.
	<i>c14to38</i>	Combine Overstruck Characters to Single Code Point.
	<i>mctoasa</i>	Convert CCW Operation Codes to ASA Carriage Control.
	<i>optcdj</i>	Generate Table Reference Character (TRC).
	<i>overstr</i>	Process Overstruck Lines.
	<i>xpndhi</i>	Expand Highlighting to Space between Words.

Built-in Programs

Figure 379 (Page 3 of 3). Filters

Rearrange record	<i>change</i> <i>insert</i> <i>retab</i> <i>space</i> <i>spec</i> <i>tokenise</i> <i>untab</i> <i>vchar</i> <i>xlate</i>	Substitute Contents of Records. Insert String in Records. Replace Runs of Blanks with Tabulate Characters. Space Words Like REXX. Rearrange Contents of Records. Tokenise Records. Replace Tabulate Characters with Blanks. Recode Characters to Different Length. Transliterate Contents of Records.
Subset	<i>substring</i>	Write substring of record.
Track	<i>ckddeblock</i> <i>trackblock</i> <i>trackdeblock</i> <i>tracksquish</i> <i>trackxpan</i>	Deblock Track Data Record. Build Track Record. Deblock Track. Squish Tracks. Unsquish Tracks.
Track	<i>trackverify</i>	Verify Track Format.

Figure 380. Gateways

	<i>dam</i> <i>deal</i> <i>elastic</i> <i>fanin</i> <i>faninany</i> <i>fanintwo</i> <i>fanout</i> <i>fanouttwo</i> <i>gate</i> <i>gather</i> <i>if</i> <i>juxtapose</i> <i>maclib</i> <i>overlay</i> <i>predselect</i> <i>structure</i> <i>synchronise</i> <i>update</i>	Pass Records Once Primed. Pass Input Records to Output Streams Round Robin. Buffer Sufficient Records to Prevent Stall. Concatenate Streams. Copy Records from Whichever Input Stream Has One. Pass Records to Primary Output Stream. Copy Records from the Primary Input Stream to All Output Streams. Copy Records from the Primary Input Stream to Both Output Streams. Pass Records Until Stopped. Copy Records From Input Streams. Process Records Conditionally. Preface Record with Marker. Generate a Macro Library from Stacked Members in a COPY File. Overlay Data from Input Streams. Control Destructive Test of Records. Manage Structure Definitions. Synchronise Records on Multiple Streams. Apply an Update File.
	<i>fillup</i> <i>fitting</i> <i>httpsplit</i> <i>threeway</i> <i>warp</i>	Pass Records To Output Streams. Source or Sink for Copipe Data. Split HTTP Data Stream. Split record three ways. Pipeline Wormhole.

Figure 381. Host command interfaces

<i>cms</i>	Issue CMS Commands, Write Response to Pipeline.
<i>command</i>	Issue TSO Commands.
<i>command</i>	Issue CMS Commands, Write Response to Pipeline.
<i>cp</i>	Issue CP Commands, Write Response to Pipeline.
<i>starmon</i>	Write Records from the *MONITOR System Service.
<i>starmsg</i>	Write Lines from a CP System Service.
<i>starsys</i>	Write Lines from a Two-way CP System Service.
<i>subcom</i>	Issue Commands to a Subcommand Environment.
<i>tso</i>	Issue TSO Commands, Write Response to Pipeline.

Figure 382. Host interfaces

<i>adrspace</i>	Manage Address Spaces.
<i>alserv</i>	Manage the Virtual Machine's Access List.
<i>mapmdisk</i>	Map Minidisks Into Data spaces.

Figure 383. Look up routines

<i>ldrtbls</i>	Resolve a Name from the CMS Loader Tables.
<i>nucext</i>	Call a Nucleus Extension.
<i>rexx</i>	Run a REXX Program to Process Data.

Figure 384. Resolvers

<i>hostbyaddr</i>	Resolve IP Address into Domain and Host Name.
<i>hostbyname</i>	Resolve a Domain Name into an IP Address.

Figure 385 (Page 1 of 2). Selection stages

	<i>unique</i>	Discard or Retain Duplicate Lines.
Assembler files	<i>asmfind</i>	Select Statements from an Assembler File as XEDIT Find.
	<i>asmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
	<i>strasmfind</i>	Select Statements from an Assembler File as XEDIT Find.
	<i>strasmnfind</i>	Select Statements from an Assembler File as XEDIT NFind

Built-in Programs

Figure 385 (Page 2 of 2). Selection stages

Contents of record	<i>abbrev</i>	Select Records that Contain an Abbreviation of a Word in the First Positions.
	<i>all</i>	Select Lines Containing Strings (or Not).
	<i>find</i>	Select Lines by XEDIT Find Logic.
	<i>locate</i>	Select Lines that Contain a String.
	<i>nfind</i>	Select Lines by XEDIT NFind Logic.
	<i>nlocate</i>	Select Lines that Do Not Contain a String.
	<i>pick</i>	Select Lines that Satisfy a Relation.
	<i>strfind</i>	Select Lines by XEDIT Find Logic.
	<i>strnfind</i>	Select Lines by XEDIT NFind Logic.
	<i>verify</i>	Verify that Record Contains only Specified Characters.
	<i>wildcard</i>	Select Records Matching a Pattern.
Groups of records	<i>between</i>	Select Records Between Labels.
	<i>drop</i>	Discard Records from the Beginning or the End of the File.
	<i>flabel</i>	Select Records from the First One with Leading String.
	<i>inside</i>	Select Records between Labels.
	<i>notinside</i>	Select Records Not between Labels.
	<i>outside</i>	Select Records Not between Labels.
	<i>strflabel</i>	Select Records from the First One with Leading String.
	<i>strtolabel</i>	Select Records to the First One with Leading String.
	<i>strwhilelabel</i>	Select Run of Records with Leading String.
	<i>take</i>	Select Records from the Beginning or End of the File.
	<i>tolabel</i>	Select Records to the First One with Leading String.
	<i>whilelabel</i>	Select Run of Records with Leading String.

Figure 386. Service programs

	<i>configure</i>	Set and Query <i>CMS Pipelines</i> Configuration Variables.
	<i>help</i>	Display Help for <i>CMS Pipelines</i> or DB2.
	<i>jeremy</i>	Write Pipeline Status to the Pipeline.
	<i>query</i>	Query <i>CMS Pipelines</i> .
	<i>warplist</i>	List Wormholes.

Figure 387. Sorters

	<i>collate</i>	Collate Streams.
	<i>dfsor</i>	Interface to DFSORT/CMS.
	<i>lookup</i>	Find Records in a Reference Using a Key Field.
	<i>merge</i>	Merge Streams.
	<i>sort</i>	Order Records.

<—Read a File

< is the generic name for a device driver that reads files into the pipeline.

Depending on the operating system and the actual syntax of the parameters, < selects the appropriate device driver to perform the actual I/O to the file.

▶▶<—*string*—◀◀

Type: Device driver.

Placement: < must be a first stage.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Operating System	Minimum Release	Driver Used	Further Tests
CMS	2.1.0	< <i>oe</i>	A single word, a quoted string, or a word beginning BFS= or HFS=. A syntax error is reported if OpenExtensions is not present in the system. Either single or double quotes may be used.
	1.2.0	< <i>sfs</i>	Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
	(any)	< <i>mfsk</i>	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.
MVS	5.1.0	< <i>oe</i>	A word that contains a forward slash (/) or is enclosed in double quotes or the word must have the prefix BFS= or HFS=. OpenExtensions must be present in the system.
	(any)	< <i>mvs</i>	Other formats.

Commit Level: Refer to the appropriate device driver.

Premature Termination: Refer to the appropriate device driver.

See Also: <*mfsk*, <*mvs*, <*oe*, and <*sfs*.

Examples: Refer to the appropriate device driver.

<mdsk

Notes:

1. Use <sfs to access a file using a NAMEDEF that would be scanned by < as a mode letter or a mode letter followed by a digit.

<mdsk—Read a CMS File from a Mode

<mdsk reads a CMS file from storage, from a minidisk, or from a Shared File System (SFS) directory that has been accessed with a mode letter. The file must exist.



Type: Device driver.

Placement: <mdsk must be a first stage.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be read. A file mode or an asterisk is optional; the default is to search all modes. STATE is used to locate the file when three words are specified; EXECSTAT is used when two words are specified. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried.

Operation: If EXECSTAT is used to locate the file and the return code is 0 (indicating that the file is storage resident), the use count and recursion count are incremented before the file is accessed through the file block provided by EXECSTAT.

Reading begins at the first record in the file and continues to end-of-file.

When a file is read from disk, the file is closed before <mdsk terminates. When an storage file is read, the recursion count is decremented before <mdsk terminates.

Premature Termination: <mdsk terminates when it discovers that its output stream is not connected.

See Also: *disk*, *diskback*, *diskrandom*, *diskslow*, *members*, and *pdsdirect*.

Examples:

```
/* Read a file and count the number of words */
'pipe < input file | count words | console'
```

Notes:

1. Use *diskslow* if <mdsk fails to operate.
2. Use *diskslow* to begin to read from a particular record. Use *diskrandom* to read a particular range of records or to read records that are not sequential. (To read many records from near the beginning of a large file it may, however, be more efficient to use *drop* and *take* with <mdsk to select the range of records desired.)
3. Use *disk* or *diskslow* to treat a file that does not exist as one with no records, rather than issue a message about a missing file.
4. Use an asterisk as the third word of the argument string to bypass the search for EXECLOADED files.

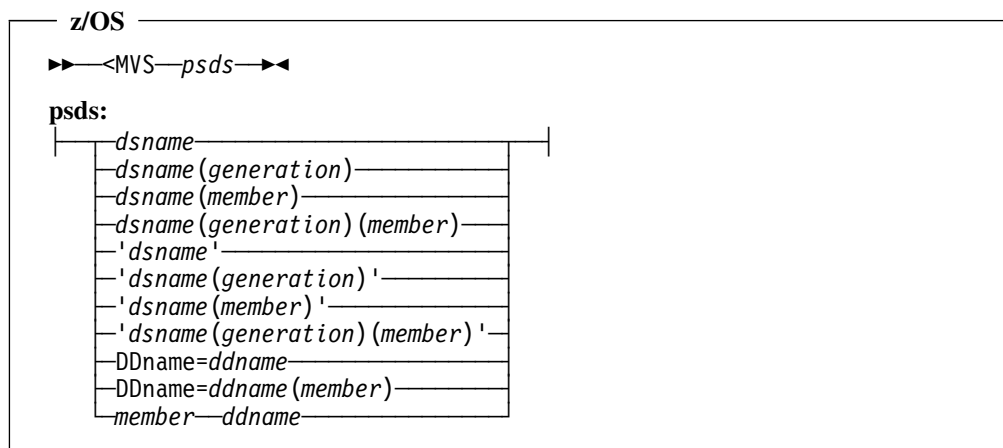
5. <mvs may obtain several records from CMS at a time. It is unspecified how many records <mvs buffers, as well as the conditions under which it does so.
6. EXECSTAT resolves a file as follows:
 - a. It searches the directory of files that are loaded with EXECLOAD or are in a logical segment (and are identified by an EXEC record) that has been attached with SEGMENT LOAD.
 - b. It searches minidisks and SFS directories, if any, ahead of the installation segment.
 - c. It searches the installation segment, if attached to the virtual machine.
 - d. It searches remaining accessed mode letters, if any.
7. The fast interface to the file system is bypassed if the bit X'10' is on in offset X'3D' of the FST that is exposed by the FSSTATE macro. Products that compress files on the fly or in other ways intercept the file system macros should turn on this bit to ensure that *CMS Pipelines* uses documented interfaces only.

Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, <mvs is transparent to return codes from CMS. Refer to the return codes for the FSREAD macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 20 The file name or file type contains an invalid character.
- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

<mvs—Read a Physical Sequential Data Set or a Member of a Partitioned Data Set

<mvs reads the contents of a physical sequential data set or a member of a partitioned data set into the pipeline. The data set must be cataloged if its data set name (DSNAME) is specified.



Type: Device driver.

Placement: <mvs must be a first stage.

Syntax Description: The data set may be specified by DSNAME, by DDNAME, or by two words specifying member name and DDNAME, respectively.

Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the DSNAME without quotes to have the prefix, if any, applied. Append paren-

<oe

theses containing a signed number to specify a relative generation of a data set that is a member of a generation data group.

To read from an already allocated data set, specify the keyword DDNAME= followed by the DDNAME already allocated. The minimum abbreviation is DD=.

A member is specified in parentheses after the DSNNAME or DDNAME. The closing right parenthesis is optional.

The third form (two blank-delimited words) can be used to read a member of an already allocated data set. The first word specifies the member name. The second word specifies the DDNAME; a leading DDNAME= keyword is optional for the second word.

The DSNNAME, DDNAME, and member names are translated to upper case.

Operation: A temporary allocation is made to process a file specified by DSNNAME. The allocation specifies the disposition DISP=(SHR,KEEP). The temporary allocation is freed when the data set or member has been processed.

Commit Level: <mvs starts on commit level -2000000000. It then opens the DCB and commits to level 0. The DCB is closed without reading if the commit return code is nonzero.

Premature Termination: <mvs terminates when it discovers that its output stream is not connected.

See Also: *listpds* and *members*.

Examples: To display a member of a procedure library:

```
pipe < 'dpmvs.logon.proclib($tsjohn)' | console  
pipe < tso.filters(copy) | console
```

The second command above could read from the data set DPJOHN.TSO.FILTERS.

To display the number of records in a member of an already allocated data set:

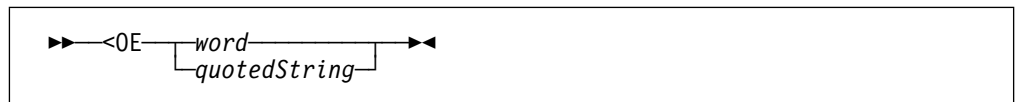
```
pipe < dd=pipehelp(count) | unpack | count lines | console  
pipe < count pipehelp | unpack | count lines | console  
pipe < count ddname=pipehelp | unpack | count lines | console
```

To read a sequential data set:

```
pipe < gotten.data | count lines | console
```

<oe—Read an OpenExtensions Text File

<oe reads a file that is stored in the OpenExtensions file system and deblocks it at line end characters (X'15'). If the file contains no line end character, a single record is written containing the entire file.



Type: Device driver.

Placement: <oe must be a first stage.

Syntax Description: A word or a quoted string is required.

Operation: <oe uses a subroutine pipeline that contains hfs to read the file and deblock TEXTFILE to perform the deblocking.

Commit Level: <oe starts on commit level -2000000000. It issues the subroutine pipeline, which will commit to 0.

Premature Termination: <oe terminates when it discovers that its output stream is not connected.

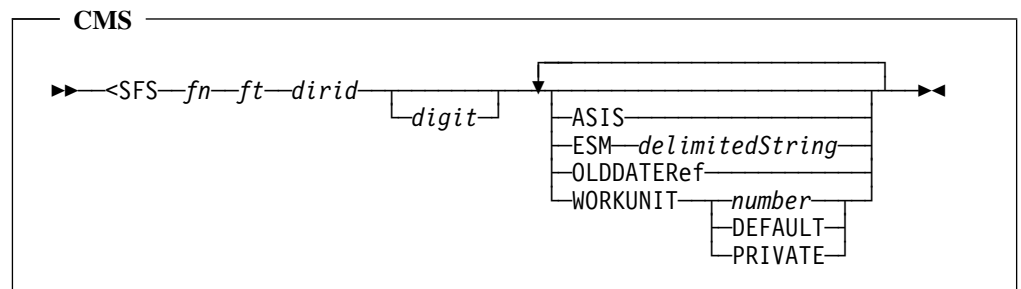
See Also: hfs.

Examples: To read a file in the current working directory:

```
pipe < .profile | count lines | console
```

<sfs—Read an SFS File

<sfs reads a file that is stored in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode.



Type: Device driver.

Placement: <sfs must be a first stage.

Syntax Description:

- fn* Specify the file name for the file.
- ft* Specify the file type for the file.
- dirid* Specify the mode, the directory, or a NAMEDEF for the directory for the file.
- digit* Specify the file mode number for the file.
- ASIS Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified.
- ESM Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
- OLDDATEREF Pass the keyword to the open routine. CMS will not update the date of last reference for the file.

<sfsslow

WORKUNIT Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is DEFAULT.

Operation: When the directory is omitted, <sfs looks for the file on all accessed modes.

Reading begins at the first record in the file and continues to end-of-file. The file is closed before <sfs terminates.

Commit Level: <sfs starts on commit level -2000000000. It creates a private unit of work if WORKUNIT PRIVATE is specified, opens the file, allocates a buffer if required, and then commits to level 0.

Premature Termination: <sfs terminates when it discovers that its output stream is not connected.

See Also: disk, diskback, diskrandom, diskslow, filetoken, members, and pdsdirect.

Examples: To read a file:
pipe < profile exec . | ...

This reads your profile from your root directory in the current file pool. < selects <sfs to process the file, because the third word is present, but does not specify a mode.

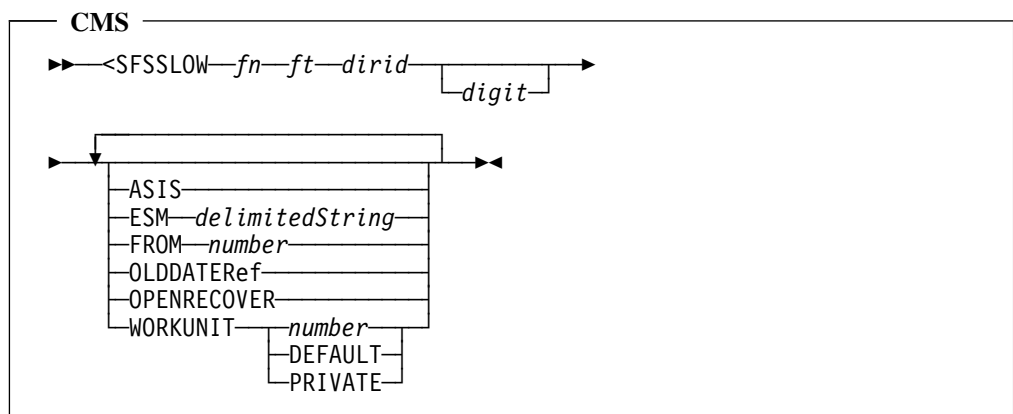
Notes:

- 1. <sfs uses the DMSVALDT callable service to resolve the actual file mode or directory name.

<sfsslow—Read an SFS File

<sfsslow reads a file that is stored in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode.

<sfsslow reads one record at a time from the host interface; it does not attempt to block reads.



Type: Device driver.

Placement: <sfsslow must be a first stage.

Syntax Description:

<i>fn</i>	Specify the file name for the file.
<i>ft</i>	Specify the file type for the file.
<i>dirid</i>	Specify the mode, the directory, or a NAMEDEF for the directory for the file.
<i>digit</i>	Specify the file mode number for the file.
ASIS	Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified.
ESM	Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
FROM	Specify the number of the first record to read. The number must be positive and must be smaller than or equal to the file size.
OLDDATeref	Pass the keyword to the open routine. CMS will not update the date of last reference for the file.
OPENRECOVER	Pass the keyword to the open routine. This particular operation is performed as if the file's attributes were RECOVER and NOTINPLACE.
WORKUNIT	Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is DEFAULT.

Operation: Unless FROM is specified, reading begins at the first record in the file. The file is closed before <sfsslow terminates.

Commit Level: <sfsslow starts on commit level -2000000000. It creates a private unit of work if WORKUNIT PRIVATE is specified, opens the file, allocates a buffer if required, and then commits to level 0.

Premature Termination: <sfsslow terminates when it discovers that its output stream is not connected.

See Also: *disk, diskback, diskrandom, filetoken, members, and pdsdirect.*

Examples: To read a file:

```
pipe <sfsslow profile exec . | ...
```

This reads your profile from your top directory in the current file pool.

>

>—Replace or Create a File

> is the generic name for a device driver that replaces files with data in the pipeline.

Depending on the operating system and the actual syntax of the parameters, > selects the appropriate device driver to perform the actual I/O to the file.

▶▶>—*string*—◀◀

Type: Device driver.

Placement: > must not be a first stage.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Operating System	Minimum Release	Driver Used	Further Tests
CMS	2.1.0	> <i>oe</i>	A single word, a quoted string, or a word beginning BFS= or HFS=. A syntax error is reported if OpenExtensions is not present in the system. Either single or double quotes may be used.
	1.2.0	> <i>sfs</i>	Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
	(any)	> <i>mnsk</i>	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.
MVS	5.1.0	> <i>oe</i>	A word that contains a forward slash (/) or is enclosed in double quotes or the word must have the prefix BFS= or HFS=. OpenExtensions must be present in the system.
	(any)	> <i>mvs</i>	Other formats.

Record Delay: > strictly does not delay the record.

Commit Level: Refer to the appropriate device driver.

Premature Termination: Refer to the appropriate device driver.

See Also: >*mnsk*, >*mvs*, >*oe*, and >*sfs*.

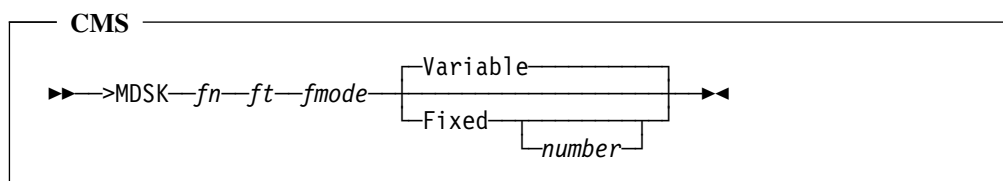
Examples: Refer to the appropriate device driver.

Notes:

1. Use >sfs to access a file using a NAMEDEF that would be scanned by > as a mode letter or a mode letter followed by a digit.
2. >sfs maintains authorisations and other attributes for the file when it is replaced, whereas >mnsk creates a work file and as a result loses such information.

>mnsk—Replace or Create a CMS File on a Mode

>mnsk replaces a file on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter. A file is created if one does not exist.



Type: Device driver.

Placement: >mnsk must not be a first stage.

Syntax Description: Specify as blank-delimited words the name, type, and mode of the file to be created. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried. Append a mode number to the mode letter to create a file with this particular mode number. The mode number of an existing file is retained when you specify a mode letter without a number; the default is 1 if the file does not exist. The optional arguments designate the file format of the file that is created. VARIABLE specifies a file that has record format variable. FIXED creates a file that has record format fixed. An additional number (the record length) may be specified for such a file. When the record length is specified, it is ensured that all input records have that particular length. When the number is omitted, the first record that is not null determines the record length of the file. The default record format is VARIABLE.

Operation: If a file already exists with the name specified, a utility file is written. When this file has been written successfully, the original file is erased and the utility file renamed to the specified name. An existing file is erased if there are no records containing data on any input stream. The file is closed before >mnsk terminates.

Streams Used: >mnsk first creates the file from records on the primary input stream that are not null; all input records are also copied to the primary output stream. The primary output stream is severed at end-of-file on the primary input stream. The first records of the file are then overwritten with any records from the secondary input stream that are not null. All records from the secondary input stream are copied to the secondary output stream after they are written to the file.

Warning: When the secondary input stream is connected, records read from it must have the same length as the records they replace in the file, but this is not enforced by CMS for variable record format files; CMS truncates a variable record format file without indication of error if a record is replaced with one of different length, be that shorter or longer.

Record Delay: >mnsk strictly does not delay the record.

>*m*disk

See Also: >>, *disk*, *diskslow*, and *diskupdate*.

Examples:

```
! /* Create a file with a single line in it */  
! 'pipe literal This is a single line.| >mdisk one liner a'
```

Notes:

1. Use *diskslow* if >*m*disk fails to operate.
2. Null input records are copied to the output (if connected), but not to the file; CMS files cannot contain null records.
3. An asterisk (*) cannot be specified as the file mode.
4. If the existing file is large and not needed to create the new one, it should be erased prior to running the pipeline so that the disk space is available to create the new file.
5. The record format of an existing fixed format file is not retained by default. Use *state* to determine the record format of a file and supply the fourth word of the result as the file format option.
6. When it is processing records from the primary input stream, >*m*disk may deliver several records at a time to CMS to improve performance. The file may not be in its eventual format while it is being created; it should not be accessed (by any means) before >*m*disk terminates. It is unspecified how many records >*m*disk buffers, as well as the conditions under which it does so.
7. When a file is replaced, the new contents will not be visible before >*m*disk terminates.
8. On CMS9 and later releases, use >*sfs* to replace files in SFS. You can accomplish this either by specifying >*sfs* explicitly or by specifying a directory with >.
9. Connect the secondary input stream when creating CMS libraries or packed files where the first record has a pointer to the directory or contains the unpacked record length of a packed file. The stage that generates the file (for instance, *maclib*) can write a placeholder first record on the primary output stream initially; it then writes the real first record to a stream connected to the secondary input stream of >*m*disk when the complete file has been processed and the location and size of the directory are known.
10. The fast interface to the file system is bypassed if the bit X'10' is on in offset X'3D' of the FST that is exposed by the FSSTATE macro. Products that compress files on the fly or in other ways intercept the file system macros should turn on this bit to ensure that *CMS Pipelines* uses documented interfaces only.
11. An existing file is not rewritten in its place, even when mode number 6 is specified or the existing file has mode number 6. (Use *diskupdate* instead.)
12. Use >*sfs* with ASIS to create a minidisk file that has a mixed case file name or file type, or both.

Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, >*m*disk is transparent to return codes from CMS. Refer to the return codes for the FSWRITE macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 1 You do not have write authority to the file.
- 13 The disk is full.
- 16 Conflict when writing a buffer; this indicates that a file with the same name has been created by another stage.
- 20 The file name or file type contains an invalid character.

- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

Configuration Variables: Two configuration variables govern how >mdsk replaces an existing file in an SFS directory. In all cases, >mdsk creates a temporary file; the file name and file type are controlled by the DISKTEMPFILETYPE configuration variable; the DISKREPLACE configuration variable controls how the new file replaces the old one.

The configuration variable DISKTEMPFILETYPE governs how >mdsk creates the file name and the file type for the temporary file when it replaces an existing file that resides in an SFS directory.

TOD : :	The file name and file type are the unpacked hexadecimal value of the time-of-day clock at the time the temporary file is created. This creates a unique temporary file across a processor complex, so long as all participants observe the protocol.
CMSUT1	The file name is made unique within the virtual machine. The file type is CMSUT1. CMSUT1 is the default in all styles.
USERID	The file name is made unique within the virtual machine. The file type is the user ID as reported by diagnose 0. This creates a unique temporary file across a system.

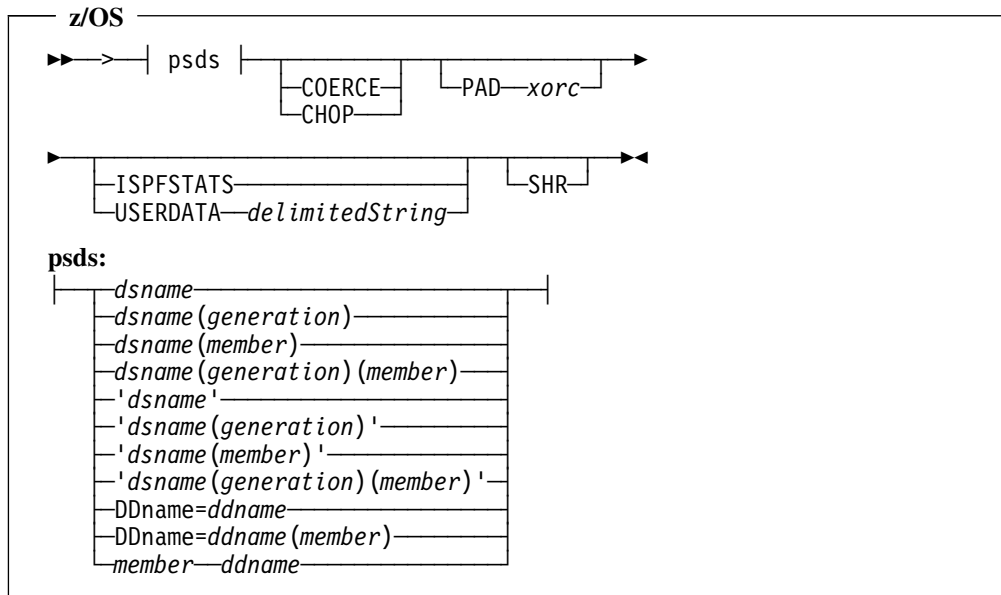
The configuration variable DISKREPLACE governs how >mdsk replaces an existing file that resides in an SFS directory.

COPY	>mdsk performs a copy operation in the SFS server to replace the contents of the file with the contents of the temporary file it first created. File authorisations and creation date will remain unchanged. COPY is the default in the DMS style.
REPLACE	>mdsk first creates a temporary file. It then erases the existing file and renames the temporary file to the file name. This changes the creation date for the file and drops all authorisations. REPLACE is the default in the PIP and FPL styles.

>mvs—Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set

>mvs rewrites a physical sequential data set or replaces a member of a partitioned data set. A new member is created if the specified member does not already exist in the data set. The data set must be cataloged if a data set name (DSNAME) is specified.

>mvS



Type: Device driver.

Placement: >mvS must not be a first stage.

Syntax Description: The data set may be specified by DSNAME or by DDNAME.

Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the DSNAME without quotes to have the prefix, if any, applied. Append parentheses containing a signed number to specify a relative generation of a data set that is a member of a generation data group.

To rewrite an already allocated data set, specify the keyword DDNAME= followed by the DDNAME already allocated. The minimum abbreviation is DD=.

A member is specified in parentheses after the DSNAME or DDNAME. The closing right parenthesis is optional.

The third form (two blank-delimited words) can be used to write a member of an already allocated data set. The first word specifies the member name. The second word specifies the DDNAME; a leading DDNAME= keyword is optional for the second word.

The DSNAME, DDNAME, and member names are translated to upper case.

The options COERCE, CHOP, and PAD are used with fixed record format data sets. COERCE specifies that the input records should be padded with blanks or truncated to the record length of the data set. CHOP specifies that long records are truncated; input records must be at least as long as the record length for the data set. PAD specifies the pad character to use when padding the record. Input records must not be longer than the record length of the data set when PAD is specified alone.

ISPFSTAT or USERDATA may be specified for a partitioned data set. Specify USERDATA to insert literal user data in the directory entry. Specify ISPFSTAT to make >mvS update or create user data in the format maintained by ISPF.

Specify SHR to allocate the data set shared. The default is DISP=OLD.

Operation: A temporary allocation is made to process a file specified by DSNNAME. The allocation specifies the disposition DISP=(OLD,KEEP) unless SHR is specified. The temporary allocation is freed when the data set has been processed.

Streams Used: >mvS passes the input to the output.

Record Delay: >mvS strictly does not delay the record.

Commit Level: >mvS starts on commit level -2000000000. It allocates the data set (if required) and then commits to 0. The data set is not opened if the commit return code is nonzero. The data set is opened on commit level 0.

See Also: >>.

Examples: To replace (or create) a member of a procedure library:
pipe literal /**/ 'output Hello' | > tso.filters(hello)

Notes:

1. Do not replace two members in a partitioned data set concurrently; z/OS does not support this.
2. Use *readpds* to read members whose names are not upper case alphanumerics.
3. The installation can set ISPFSTAT as the default. It can also select one of the coercing options as the default. Refer to the installation instructions.
4. Specifying SHR only affects the allocation; the user must ensure the integrity of the data set. In particular, specifying SHR does not imply support for concurrent update of members in a partitioned data set.

>oe—Replace or Create an OpenExtensions Text File

>oe replaces the contents of a text file that is stored in the OpenExtensions file system. It creates the file if it does not exist.

>oe appends a line end character (X'15') to each input record before it writes the record to the file.



Type: Device driver.

Placement: >oe must not be a first stage.

Syntax Description: A word or a quoted string is required.

Operation: >oe uses a subroutine pipeline that contains *block* TEXTFILE to append the line end and *hfsreplace* to replace the file. When the file exists, >oe buffers the new contents of the file and thus replaces the specified file when it reaches end-of-file.

Record Delay: >oe strictly does not delay the record.

Commit Level: >oe starts on commit level -2000000000. It issues the subroutine pipeline, which will commit to 0.

>sfs

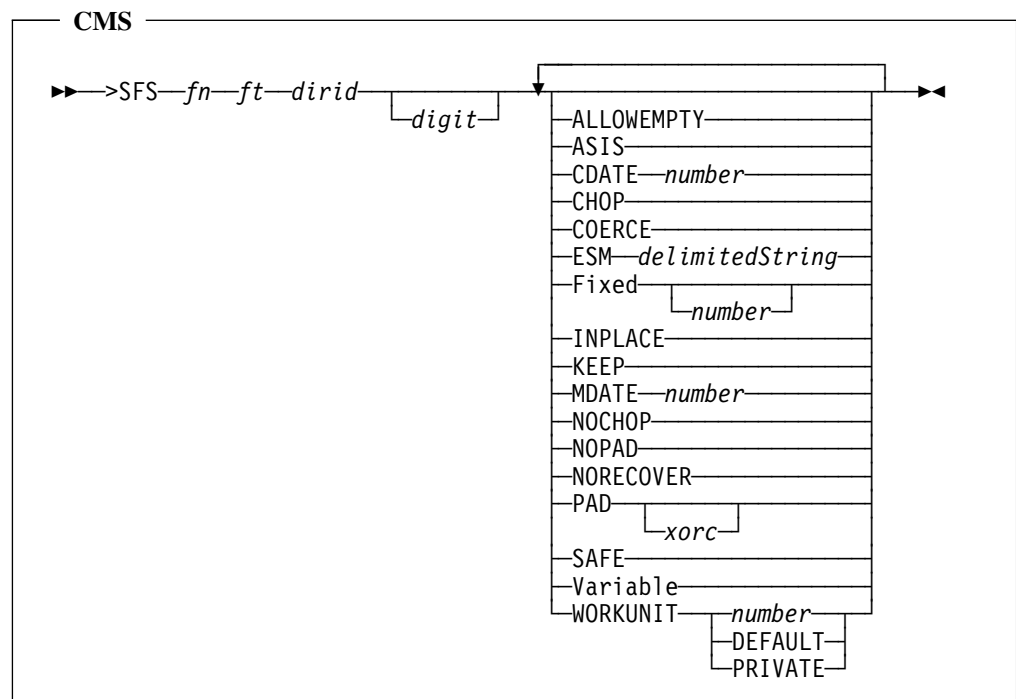
See Also: *hfsreplace*.

Examples: To replace a file in the current working directory:

```
pipe literal line two | literal line one | >oe 'two line file'
```

>sfs—Replace or Create an SFS File

>sfs replaces a file in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode. A file is created if one does not exist.



Type: Device driver.

Placement: >sfs must not be a first stage.

Syntax Description:

- | | |
|--------------|--|
| <i>fn</i> | Specify the file name for the file. |
| <i>ft</i> | Specify the file type for the file. |
| <i>dirid</i> | Specify the mode, the directory, or a NAMEDEF for the directory for the file. |
| <i>digit</i> | Specify the file mode number for the file. |
| ALLOWEMPTY | Pass the keyword to the open routine. When ALLOWEMPTY is specified, CMS creates an empty file if no input is read. The default is not to create a file when there is no input. |
| ASIS | Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified. |

C	CDATE	Specify the file creation date and time. The timestamp contains eight to fourteen digits. The first eight digits specify the year (four digits), the month (two digits), and the day (two digits). The remaining digits are padded on the right with zeros to form six digits time consisting of the hour, the minute, and the second. A twenty-four hour clock is used.
	CHOP	Truncate long input records to the logical record length of the file. The logical record length of a variable record format file is 65535 bytes unless a smaller value is specified after the VARIABLE keyword.
	COERCE	A convenience for PAD CHOP.
	ESM	Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
	FIXED	The record length may be specified after FIXED. Create a new file with fixed record format; verify that an existing file has fixed record format. If the record length is specified and the file exists, it is verified that the file is of the specified record length.
	INPLACE	Pass the keyword to the open routine. The file will be updated in place.
!	KEEP	KEEP is ignored unless WORKUNIT PRIVATE is specified or defaulted. When KEEP is specified, changes are committed to the file even when an error has occurred. The default is to roll back the unit of work. KEEP is mutually exclusive with SAFE.
	MDATE	Specify the file modification date and time. The timestamp contains eight to fourteen digits. The first eight digits specify the year (four digits), the month (two digits), and the day (two digits). The remaining digits are padded on the right with zeros to form six digits time consisting of the hour, the minute, and the second. A twenty-four hour clock is used.
	NOCHOP	Do not truncate long records. Issue a message instead.
:	NOPAD	Do not pad short records. For files that have fixed record format, issue a message and terminate when an input record is shorter than the record length; ignore null records when writing files that have variable record format (but pass the null record to the primary output stream).
:		
:		
:		
	NORECOVER	Pass the keyword to the open routine. Changes to the file may persist after the unit of work is rolled back.
	PAD	Pad short records with the character specified. The blank is used as the pad character if the following word does not scan as an <i>xorc</i> . Pad short records on the right to the file's record length in a file that has fixed record format. Write a single pad character for a null input record in a file that has variable record format. In both cases, pass the unmodified input record to the primary output stream.
:		
:		
!	SAFE	SAFE is rejected if WORKUNIT PRIVATE is neither specified nor defaulted. When SAFE is specified, >sf/s performs a pipeline commit to level 1 before it returns the unit of work. It rolls back the unit of work if the commit does not complete with return code 0. SAFE is mutually exclusive with KEEP.
	VARIABLE	The record length may be specified after VARIABLE. Create a new variable record format file; verify that an existing file has variable record format.

>>

WORKUNIT Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is PRIVATE.

Streams Used: >sfs first creates the file from records on the primary input stream that are not null; all input records are also copied to the primary output stream. The primary output stream is severed at end-of-file on the primary input stream. The first records of the file are then overwritten with any records from the secondary input stream that are not null. All records from the secondary input stream are copied to the secondary output stream after they are written to the file. >sfs terminates with an error message if a record is replaced with one of a different length.

Record Delay: >sfs strictly does not delay the record.

Commit Level: >sfs starts on commit level -2000000000. It creates a private unit of work if WORKUNIT PRIVATE is specified or defaulted, opens the file, allocates a buffer if required, and then commits to level 0.

See Also: >>, disk, diskslow, diskupdate, and filetoken.

Examples: To create a file that contains a single line in the root directory:

```
pipe literal one line | >sfs one liner .
```

>>—Append to or Create a File

>> is the generic name for a device driver that appends data in the pipeline to files.

Depending on the operating system and the actual syntax of the parameters, >> selects the appropriate device driver to perform the actual I/O to the file.

▶▶>>—string▶▶

Type: Device driver.

Placement: >> must not be a first stage.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Operating System	Minimum Release	Driver Used	Further Tests
CMS	2.1.0	>>oe	A single word, a quoted string, or a word beginning BFS= or HFS=. A syntax error is reported if OpenExtensions is not present in the system. Either single or double quotes may be used.

Operating System	Minimum Release	Driver Used	Further Tests
	1.2.0	>>sfs	Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
	(any)	>>mdsk	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.
MVS	5.1.0	>>oe	A word that contains a forward slash (/) or is enclosed in double quotes or the word must have the prefix BFS= or HFS=. OpenExtensions must be present in the system.
	(any)	>>mvs	Other formats.

Record Delay: >> strictly does not delay the record.

Commit Level: Refer to the appropriate device driver.

Premature Termination: Refer to the appropriate device driver.

See Also: >>mdsk, >>mvs, >>oe, and >>sfs.

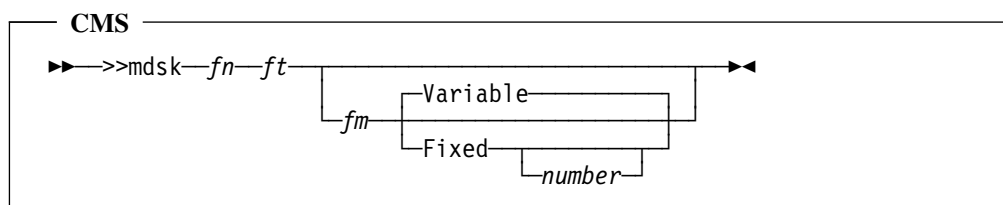
Examples: Refer to the appropriate device driver.

Notes:

1. Use >>sfs to access a file using a NAMEDEF that would be scanned by >> as a mode letter or a mode letter followed by a digit.

>>mdsk—Append to or Create a CMS File on a Mode

>>mdsk appends records to a file on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter. A file is created if one does not exist.



Type: Device driver.

Placement: >>mdsk must not be a first stage.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be appended to. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried. The file is

>>mdsk

created as A1 if no file mode (or an asterisk) is specified and no file is found with the name and type given. The record format and (for fixed format files) the record length are optional arguments. The default is the characteristics of an existing file when appending, VARIABLE when a file is being created. When the file exists, the specified record format must match the characteristics of the file.

Operation: Records are appended to an existing CMS file; a new file is created (with an upper case file name and type) if no file is found to append to. The file is closed before >>mdsk terminates.

Streams Used: >>mdsk first appends or creates the file from records on the primary input stream that are not null; all input records are also copied to the primary output stream. The primary output stream is severed at end-of-file on the primary input stream. The first records of the file are then overwritten with any records from the secondary input stream that are not null. All records from the secondary input stream are copied to the secondary output stream after they are written to the file.

Warning: When the secondary input stream is connected, records read from it must have the same length as the records they replace in the file, but this is not enforced by CMS for variable record format files; CMS truncates a variable record format file without indication of error if a record is replaced with one of different length, be that shorter or longer.

Record Delay: >>mdsk strictly does not delay the record.

See Also: >, disk, diskslow, and diskupdate.

Examples:

```
/* Append a line to a file */  
'pipe literal this is a single line|>>mdsk many liner'
```

Notes:

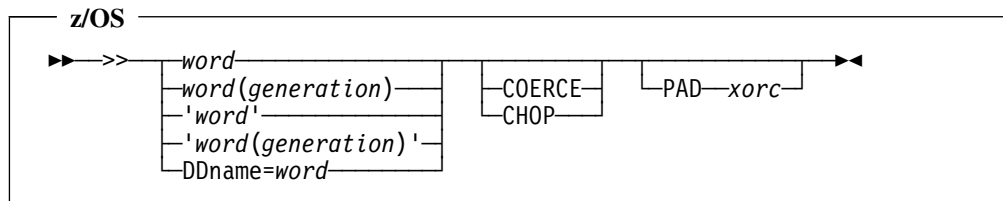
1. Use *diskslow* if >>mdsk fails to operate.
2. Null input records are copied to the output (if connected), but not to the file; CMS files cannot contain null records.
3. Use *diskslow* to begin to write at a particular record. Use *diskupdate* to replace random records.
4. When it is processing records from the primary input stream, >>mdsk may deliver several records at a time to CMS to improve performance. The file may not be in its eventual format while it is being created; it should not be accessed (by any means) before >>mdsk terminates. It is unspecified how many records >>mdsk buffers, as well as the conditions under which it does so.
5. Connect the secondary input stream when creating CMS libraries or packed files where the first record has a pointer to the directory or contains the unpacked record length of a packed file. The stage that generates the file (for instance, *maclib*) can write a placeholder first record on the primary output stream initially; it then writes the real first record to a stream connected to the secondary input stream of >>mdsk when the complete file has been processed and the location and size of the directory are known.
6. The fast interface to the file system is bypassed if the bit X'10' is on in offset X'3D' of the FST that is exposed by the FSSTATE macro. Products that compress files on the fly or in other ways intercept the file system macros should turn on this bit to ensure that *CMS Pipelines* uses documented interfaces only.

Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, >>mvs is transparent to return codes from CMS. Refer to the return codes for the FSWRITE macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 1 You do not have write authority to the file.
- 13 The disk is full.
- 16 Conflict when writing a buffer; this indicates that a file with the same name has been created by another stage.
- 20 The file name or file type contains an invalid character.
- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

>>mvs—Append to a Physical Sequential Data Set

>>mvs appends data from the pipeline to a physical sequential data set. The data set must be cataloged if its data set name (DSNAME) is specified.



Type: Device driver.

Placement: >>mvs must not be a first stage.

Syntax Description: The data set may be specified by DSNAME or by DDNAME.

Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the DSNAME without quotes to have the prefix, if any, applied. Append parentheses containing a signed number to specify a relative generation of a data set that is a member of a generation data group.

To append to an already allocated data set, specify the keyword DDNAME= followed by the DDNAME already allocated. The minimum abbreviation is DD=.

The options COERCE, CHOP, and PAD are used with fixed record format data sets. COERCE specifies that the input records should be padded with blanks or truncated to the record length of the data set. CHOP specifies that long records are truncated; input records must be at least as long as the record length for the data set. PAD specifies the pad character to use when padding the record. Input records must not be longer than the record length of the data set when PAD is specified alone.

Operation: A temporary allocation is made to process a file specified by DSNAME. The allocation specifies the disposition DISP=(MOD,KEEP). The temporary allocation is freed when the data set has been processed.

Streams Used: >>mvs passes the input to the output.

Record Delay: >>mvs strictly does not delay the record.

Commit Level: >>mvb starts on commit level -2000000000. It allocates the data set (if required), opens the DCB, and commits to level 0.

See Also: >.

Examples: To append records to a log data set:

```
pipe literal I'm here... | >> tso.log
```

Notes:

1. >>mvb cannot append to a member of a partitioned data set. Use < to read the member and > to replace it.

>>oe—Append to or Create an OpenExtensions Text File

>>oe appends lines to a text file that is stored in the OpenExtensions file system. It creates the file if it does not exist.

>>oe appends a line end character (X'15') to each input record before it writes the record to the file.



Type: Device driver.

Placement: >>oe must not be a first stage.

Syntax Description: A word or a quoted string is required.

Operation: >>oe uses a subroutine pipeline that contains *block* TEXTFILE to append the line end and *hfs* to append to the file.

Record Delay: >>oe strictly does not delay the record.

Commit Level: >>oe starts on commit level -2000000000. It issues the subroutine pipeline, which will commit to 0.

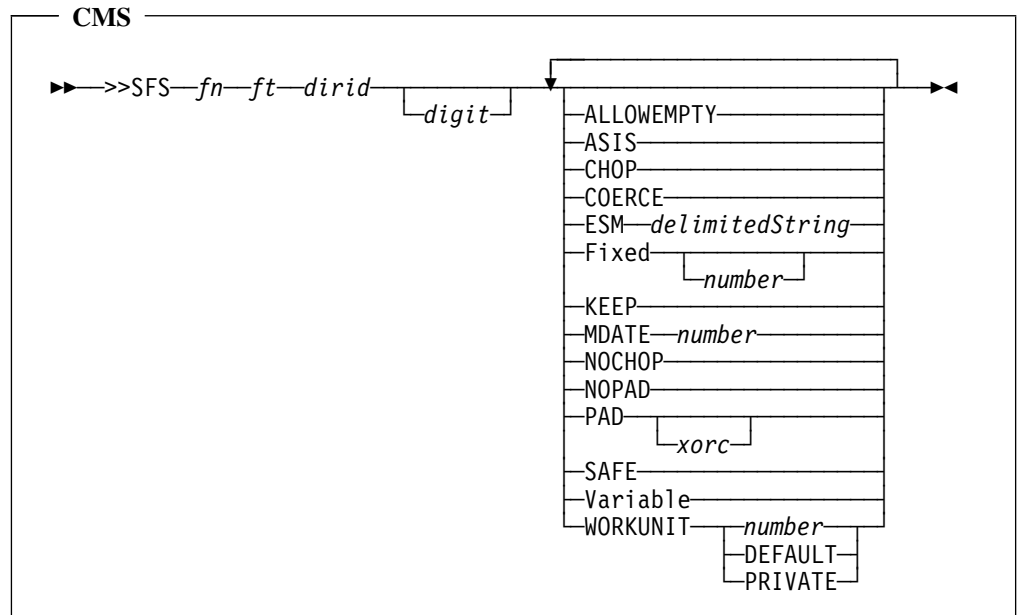
See Also: *hfsreplace*.

Examples: To append to a file in the current working directory:

```
pipe literal line four | literal line three | >>oe 'two line file'
```

>>sfs—Append to or Create an SFS File

>>sfs appends records to a file in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode. A file is created if one does not exist.



Type: Device driver.

Placement: >>sfs must not be a first stage.

Syntax Description:

<i>fn</i>	Specify the file name for the file.
<i>ft</i>	Specify the file type for the file.
<i>dirid</i>	Specify the mode, the directory, or a NAMEDEF for the directory for the file.
<i>digit</i>	Specify the file mode number for the file.
ALLOWEMPTY	Pass the keyword to the open routine. When ALLOWEMPTY is specified, CMS creates an empty file if no input is read. The default is not to create a file when there is no input.
ASIS	Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified.
CHOP	Truncate long input records to the logical record length of the file. The logical record length of a variable record format file is 65535 bytes unless a smaller value is specified after the VARIABLE keyword.
COERCE	A convenience for PAD CHOP.
ESM	Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
FIXED	The record length may be specified after FIXED. Create a new file with fixed record format; verify that an existing file has fixed record format. If the record length is specified and the file exists, it is verified that the file is of the specified record length.

>>sfs

!	KEEP	KEEP is ignored unless WORKUNIT PRIVATE is specified or defaulted. When KEEP is specified, changes are committed to the file even when an error has occurred. The default is to roll back the unit of work. KEEP is mutually exclusive with SAFE.
	MDATE	Specify the file modification date and time. The timestamp contains eight to fourteen digits. The first eight digits specify the year (four digits), the month (two digits), and the day (two digits). The remaining digits are padded on the right with zeros to form six digits time consisting of the hour, the minute, and the second. A twenty-four hour clock is used.
	NOCHOP	Do not truncate long records. Issue a message instead.
:	NOPAD	Do not pad short records. For files that have fixed record format, issue a message and terminate when an input record is shorter than the record length; ignore null records when writing files that have variable record format (but pass the null record to the primary output stream).
:		
:		
:	PAD	Pad short records with the character specified. The blank is used as the pad character if the following word does not scan as an <i>xorc</i> . Pad short records on the right to the file's record length in a file that has fixed record format. Write a single pad character for a null input record in a file that has variable record format. In both cases, pass the unmodified input record to the primary output stream.
:		
:		
!	SAFE	SAFE is rejected if WORKUNIT PRIVATE is neither specified nor defaulted. When SAFE is specified, >>sfs performs a pipeline commit to level 1 before it returns the unit of work. It rolls back the unit of work if the commit does not complete with return code 0. SAFE is mutually exclusive with KEEP.
	VARIABLE	The record length may be specified after VARIABLE. Create a new variable record format file; verify that an existing file has variable record format.
	WORKUNIT	Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is PRIVATE.

Streams Used: >>sfs first appends or creates the file from records on the primary input stream that are not null; all input records are also copied to the primary output stream. The primary output stream is severed at end-of-file on the primary input stream. The first records of the file are then overwritten with any records from the secondary input stream that are not null. All records from the secondary input stream are copied to the secondary output stream after they are written to the file. An error message is issued if a record is replaced with one of a different length.

Record Delay: >>sfs strictly does not delay the record.

Commit Level: >>sfs starts on commit level -2000000000. It creates a private unit of work if WORKUNIT PRIVATE is specified or defaulted, opens the file, allocates a buffer if required, and then commits to level 0.

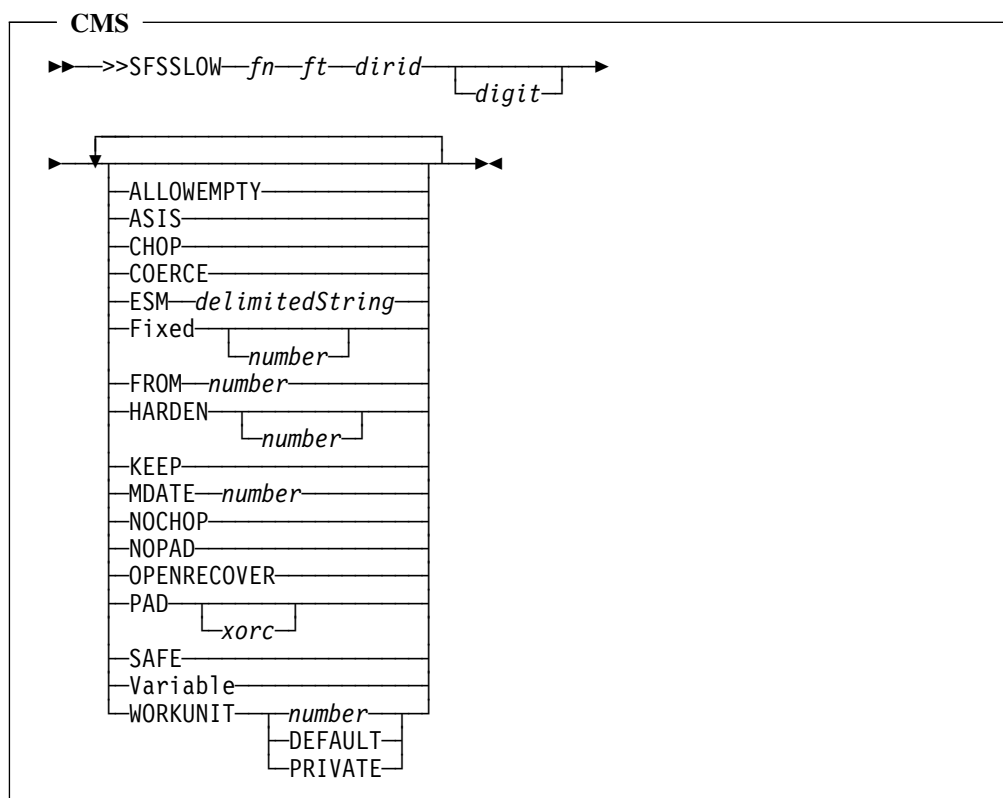
See Also: *disk*, *diskslow*, *diskupdate*, and *filetoken*.

Examples: Append a line to a file in the root directory:
 pipe literal one more line | >>sfs one liner .

>>sfsslow—Append to or Create an SFS File

>>sfsslow appends records to a file in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode. A file is created if one does not exist.

>>sfsslow writes one record at a time; it does not attempt to block writes.



Type: Device driver.

Placement: >>sfsslow must not be a first stage.

Syntax Description:

- fn* Specify the file name for the file.
- ft* Specify the file type for the file.
- dirid* Specify the mode, the directory, or a NAMEDEF for the directory for the file.
- digit* Specify the file mode number for the file.
- ALLOWEMPTY Pass the keyword to the open routine. When ALLOWEMPTY is specified, CMS creates an empty file if no input is read. The default is not to create a file when there is no input.

>>sfsslow

ASIS	Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified.
CHOP	Truncate long input records to the logical record length of the file. The logical record length of a variable record format file is 65535 bytes unless a smaller value is specified after the VARIABLE keyword.
COERCE	A convenience for PAD CHOP.
ESM	Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
FIXED	The record length may be specified after FIXED. Create a new file with fixed record format; verify that an existing file has fixed record format. If the record length is specified and the file exists, it is verified that the file is of the specified record length.
FROM	Specify the number of the first record to write. The number must be positive; it must be smaller than or equal to one plus the file size for a file that has variable record format.
HARDEN	Perform SFS commit operations to make the file contents permanent before end-of-file is read. Specify the number of records to write to the file between each SFS commit operation. HARDEN is mutually exclusive with SAFE.
KEEP	KEEP is ignored unless WORKUNIT PRIVATE is specified or defaulted. When KEEP is specified, changes are committed to the file even when an error has occurred. The default is to roll back the unit of work. KEEP is mutually exclusive with SAFE.
MDATE	Specify the file modification date and time. The timestamp contains eight to fourteen digits. The first eight digits specify the year (four digits), the month (two digits), and the day (two digits). The remaining digits are padded on the right with zeros to form six digits time consisting of the hour, the minute, and the second. A twenty-four hour clock is used.
NOCHOP	Do not truncate long records. Issue a message instead.
NOPAD	Do not pad short records. For files that have fixed record format, issue a message and terminate when an input record is shorter than the record length; ignore null records when writing files that have variable record format (but pass the null record to the primary output stream).
OPENRECOVER	Pass the keyword to the open routine. This particular operation is performed as if the file's attributes were RECOVER and NOTINPLACE.
PAD	Pad short records with the character specified. The blank is used as the pad character if the following word does not scan as a <i>xorc</i> . Pad short records on the right to the file's record length in a file that has fixed record format. Write a single pad character for a null input record in a file that has variable record format. In both cases, pass the unmodified input record to the primary output stream.

!

SAFE	SAFE is rejected if WORKUNIT PRIVATE is neither specified nor defaulted. When SAFE is specified, >>sfsslow performs a pipeline commit to level 1 before it returns the unit of work. It rolls back the unit of work if the commit does not complete with return code 0. SAFE is mutually exclusive with HARDEN and KEEP.
VARIABLE	The record length may be specified after VARIABLE. Create a new variable record format file; verify that an existing file has variable record format.
WORKUNIT	Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is PRIVATE.

Streams Used: >>sfsslow first appends or creates the file from records on the primary input stream that are not null; all input records are also copied to the primary output stream. The primary output stream is severed at end-of-file on the primary input stream. The first records of the file are then overwritten with any records from the secondary input stream that are not null. All records from the secondary input stream are copied to the secondary output stream after they are written to the file. An error message is issued if a record is replaced with one of a different length.

Record Delay: >>sfsslow strictly does not delay the record.

Commit Level: >>sfsslow starts on commit level -2000000000. It creates a private unit of work if WORKUNIT PRIVATE is specified or defaulted, opens the file, allocates a buffer if required, and then commits to level 0.

See Also: >>, disk, diskupdate, and filetoken.

Examples: Append a line to a file in the root directory:

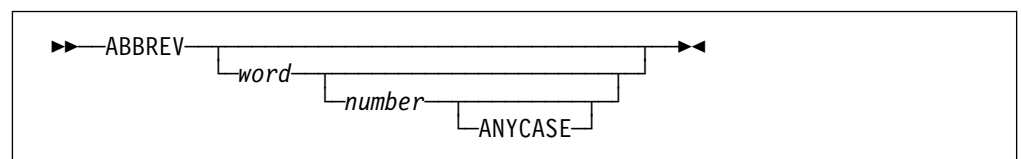
```
pipe literal one more line | >>sfsslow one liner .
```

To append to a log file and make sure the lines are immediately added to the file:

```
pipe ... | >>sfsslow log file production.logs harden 1 inplace
```

abbrev—Select Records that Contain an Abbreviation of a Word in the First Positions

abbrev selects records that contain an abbreviation of a specified word in the first positions. It discards records that do not begin with an abbreviation of the specified word or do not contain a minimum count of characters in the first blank-delimited word.



Type: Selection stage.

Syntax Description: A word, a number and a keyword are optional.

The word specifies the characters to compare against the beginning of input records. The default is a null word. The number specifies the minimum count of characters that must be present to select the record. The default is zero, which means that any abbreviation down to a null record or a leading blank will be selected. Specify ANYCASE to make the comparison case insensitive.

Operation: *abbrev* compares the leading columns of each record against the specified word until a blank or the end of the record is met. The record is passed to the primary output stream if a minimum abbreviation of the specified word is present. Otherwise, the record is discarded, or passed to the secondary output stream if the secondary output stream is connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *abbrev* strictly does not delay the record.

Commit Level: *abbrev* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *abbrev* terminates when it discovers that no output stream is connected.

Examples: To select records for “string”, with three characters minimum:

```
pipe literal st str string strings | split | abbrev string 3 | console
▶str
▶string
▶Ready;
```

Notes:

1. *abbrev* is similar to the REXX built-in function `abbrev()`.
2. Using *abbrev* without arguments selects null records and records that contain a leading blank.
3. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

acigroup—Write ACI Group for Users

acigroup obtains the ACI group from CP for user names specified in input records.



Type: Device driver.

Operation: For each word in the input, *acigroup* makes it upper case and obtains the ACI group from CP. When the user ID exists, a 16-byte record is written to the primary output stream. The record contains the 8-character user ID followed by the group. When the user ID does not exist and the secondary output stream is defined, an 8-byte record containing the user ID is written to this stream.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded.

Record Delay: *acigroup* does not delay the record.

Commit Level: *acigroup* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *acigroup* terminates when it discovers that no output stream is connected.

addrdw—Prefix Record Descriptor Word to Records

addrdw adds a record descriptor word to the beginning of each record. The type of record descriptor is specified as the argument.



Type: Filter.

Syntax Description:

VARIABLE	Prefix a record descriptor word as used by z/OS (four bytes). The record descriptor word contains the total length of the output record in the first two bytes; the next two bytes contain binary zeros.
CMS	Prefix a record descriptor word as used by CMS (two bytes). The first two bytes contain the length of the input record (unsigned).
SF	Prefix a structured field length specifier (two bytes). The first two bytes contain the length of the output record; this is two more than the length of the input record.
CMS4	Prefix a record descriptor word of four bytes. The first four bytes contain the length of the input record.
SF4	Prefix a record descriptor word of four bytes. The first four bytes contain the length of the output record; this is four more than the length of the input record.

Operation: Null records are discarded when CMS or CMS4 is specified.

Record Delay: *addrdw* strictly does not delay the record.

Premature Termination: *addrdw* terminates when it discovers that its output stream is not connected.

Converse Operation: *deblock*.

See Also: *block*.

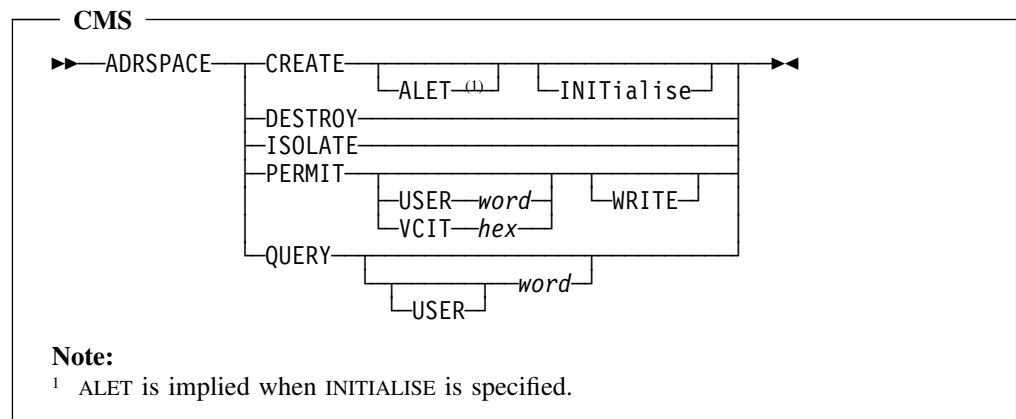
adrspace

Examples:

```
! pipe strliteral /abc/ | addrdw cms | spec 1.2 c2x 1 3-* nw | console
! ▶0003 abc
! ▶Ready;
```

adrspace—Manage Address Spaces

adrspace interfaces to the CP macro ADRSPACE to manage address spaces available to your virtual machine.



Type: Host interface.

Syntax Description:

CREATE	Create data spaces.
ALET	Also add an ALET as if the output record had been passed to ALSERV ADD WRITE. The output record includes the ALET.
INITIALISE	The first 64 bytes of the data space are initialised with information about the data space. ALET is implied when INITIALISE is specified.
DESTROY	Return a data space to VM. No additional operands are allowed.
ISOLATE	Drop all permissions to a data space. No additional operands are allowed.
PERMIT	Allow other users access to a data space. The user ID is supplied in the input record when both USER and VCIT are omitted.
USER	Specify the user name of the user to be permitted access to an address space.
VCIT	Specify virtual configuration identification token of the host-primary address space of a user.
WRITE	Permission is given to modify the address space, but still subject to storage key protection.
QUERY	Obtain ASIT and size of a data space.

!	USER	Optional keyword; required to query a data space of another user.
:	<i>word</i>	Specify the user ID of the user for whom address spaces are queried. The default is your own virtual machine.

Operation: Operation depends on the first keyword in the operands.

CREATE Data spaces are created based on the information supplied on the primary input stream. Optionally, an *ALET* is assigned. If *INITIALISE* is further specified, the first 80 bytes of the data space are set as follows (addresses in hexadecimal):

0-7	The eye-catcher 'fplasi1'.
8-F	The ASIT.
10-13	The count of pages.
14-17	Zeros. May be used as a lock.
18-1F	The user ID that created the data space.
20-38	The data space name.
38-3B	The first available byte in the data space (=X'50').
3C-3F	The last available byte in the data space (4096 times the number of pages minus 1).
40-4F	Reserved. Zeros.

The definition of this structure is built in as *fplasi1*.

DESTROY The data spaces specified by the input are destroyed.

ISOLATE The data spaces specified by the input are isolated, that is, all permissions granted for them are revoked.

PERMIT The specified user is given permission to access the data spaces specified by the input. The user is identified either by the user ID or by the address space identification token (ASIT) of the virtual machine's primary address space, also known as the virtual configuration identification token (VCIT).

QUERY Obtain the address space identification token of address spaces owned by the specified user or yourself.

Input Record Format:

CREATE The input must contain two or three blank-delimited words:

1. The name to be given to the address space.
2. The size of the address space in pages. As address spaces are complete segments, the number is rounded up by CP to the next higher multiple of 256.
3. Optionally the storage key to be assigned, the default being zero. This is specified as a two-digit hexadecimal number. The first digit specifies the storage key. The top bit of the second digit, when on, specifies that the address space is fetch protected; the three right-most bits are ignored.

adrspace

:	DESTROY	If USER or VCIT is specified with PERMIT, the record may be eight,
:	ISOLATE	twelve, or sixteen bytes; the first eight bytes are the address space
:	PERMIT	identification token (ASIT) of the data space for which the user is granted
:		access. Otherwise, the input record must be sixteen bytes and contain
:		eight bytes ASIT followed by eight bytes user ID.
:	QUERY	The name of the address space to query.

Output Record Format: The table below shows the information written to the primary output stream. When the secondary output stream is defined, the input record is passed to that stream if the referenced object does not exist.

!	CREATE	An eight byte address space identification token (ASIT) followed by a
:		four byte binary count of the number of pages available in the data
:		space, rounded up to the next multiple of 256 to indicate the actual
:		capacity available. When ALET is specified, a four bytes ALET is
:		appended to the record making it sixteen bytes in all.
:	DESTROY	The input record is passed unchanged.
:	ISOLATE	
:	PERMIT	
!	QUERY	An eight byte address space identification token (ASIT) followed by a
:		four byte binary count of the number of pages available in the data
:		space. This page count is a multiple of 256.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected. Null input records are discarded.

Record Delay: *adrspace* does not delay the record.

Commit Level: *adrspace* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *adrspace* terminates when it discovers that no output stream is connected.

See Also: *alserv* and *mapmdisk*.

Examples: See Chapter 18, "Using VM Data Spaces with *CMS Pipelines*" on page 210.

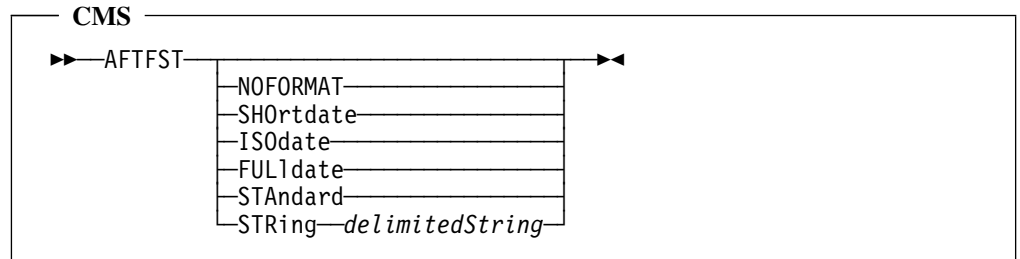
Notes:

1. *addrspace* is a synonym for *adrspace*.
2. The virtual machine must be in XC mode (this excludes z/CMS).
3. All address spaces created are destroyed by an IPL of the virtual machine.
4. Permissions granted are revoked by IPL of either of the two virtual machines.
5. Use key X'E0' on *adrspace* CREATE to interoperate with CMS programs running in user key, such as *CMS Pipelines*.

Publications: z/VM: CP Programming Services.

aftfst—Write Information about Open Files

aftfst writes a line of file status information for each CMS file that is open on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter.



Type: Arcane device driver.

Placement: *aftfst* must be a first stage.

Syntax Description: An optional keyword indicates how the output is to be formatted. The default is SHORTDATE.

NOFORMAT	The file status information is not formatted. The output record is sixty-four bytes.
FULLDATE	The file's timestamp is formatted in the American format, with the century: 3/09/1946 23:59:59.
ISODATE	The file's timestamp is formatted with the century in one of the formats approved by the International Standardisation Organisation: 1946-03-09 23:59:59.
SHORTDATE	The file's timestamp is formatted in the American format, without the century: 3/09/46 23:59:59.
STANDARD	The file's timestamp is formatted as a single word in a form that can be used for comparisons: 19460309235959.
STRING	Specify custom timestamp formatting, similar to the POSIX <code>strftime()</code> function. The delimited string specifies formatting as literal text and substitutions are indicated by a percentage symbol (%) followed by a character that defines the substitution. These substitution strings are recognised by <i>aftfst</i> : <ul style="list-style-type: none"> % A single %. %Y Four digits year including century (0000-9999). %y Two-digit year of century (00-99). %m Two-digit month (01-12). %n Two-digit month with initial zero changed to blank (1-12). %d Two-digit day of month (01-31). %e Two-digit day of month with initial zero changed to blank (1-31). %H Hour, 24-hour clock (00-23). %k Hour, 24-hour clock first leading zero blank (0-23). %M Minute (00-59). %S Second (00-60). %F Equivalent to %Y-%m-%d (the ISO 8601 date format). %T Short for %H:%M:%S. %t Tens and hundredth of a second (00-99).

Operation: A line is written for each file in the Active File Table (AFT).

Output Record Format: When NOFORMAT is specified, the output record contains 64 bytes in the format defined by the FSTD data area.

Otherwise, selected fields of the file status are formatted and written as a record: the file name, type, and mode; the record format and logical record length; the number of records and the number of disk blocks in the file; the date and time of last change to the file.

Premature Termination: *afbst* terminates when it discovers that its output stream is not connected.

See Also: *fmbfst*, *state*, and *statew*.

Examples: To issue a message when there are open files:

```
/* Write a message about unclosed files */
'PIPE',
  'aftfst noformat |',
  'count lines |',
  'nfind 0 |',
  'spec 1-* 1 / open files./ next |',
  'console'
```

Notes:

1. CMS adds an entry to the list of open files if the output from *afbst* is written to disk later in the pipeline. Buffer the output from *afbst* (for instance with *buffer* or *sort*) to ensure that you get a consistent snapshot of the file status.
2. Running this device driver may alert authors of shells that they have forgotten to close open files.
3. A file is in the AFT when it has been opened by an explicit FSOPEN or by an implicit open on the first I/O operation to the file. FSCLOSE (or the CMS command FINIS) closes the file and removes information about the file from the AFT.
4. *afbst* does not write information about files in the Shared File System (SFS) that are opened by a call to the Callable Services Library (CSL) routine DMSOPEN (or similar).
5. Be sure to set numeric digits 14 when performing comparisons on STANDARD timestamps; REXX will by default use just nine digits precision. This means that the first digit of the hour will be the least significant one and the remainder of the precision will be lost.
6. SORTED is a synonym for STANDARD.

aggrc—Compute Aggregate Return Code

aggrc reads input records, which must contain a single number each, and interprets those numbers as return codes to be aggregated in the *CMS Pipelines* style. When it has read all its input, it produces a single record that contains the aggregate return code for the numbers read. If any number is negative, the aggregate is the lowest number; otherwise it is the largest number. If *aggrc* is the last stage of a pipeline, it sets its own return code from the aggregate it has computed.

▶▶—AGGRC—▶▶

Type: Filter.

Input Record Format: One number in the interval -2147483648 to 2147483647. A leading plus sign is ignored.

Output Record Format: A number. Zero and positive numbers have no sign.

Record Delay: *aggrc* delays all records until end-of-file.

Examples:

```
pipe literal 0 99 3 6 | split | aggrc | console
▶99
▶Ready;
```

```
pipe literal 0 99 -3 6 | split | aggrc | console
▶-3
▶Ready;
```

```
pipe (listerr) literal 6 | aggrc
▶Stage returned with return code 6
▶... Issued from stage 2 of pipeline 1
▶... Running "aggrc"
▶Ready(00006);
```

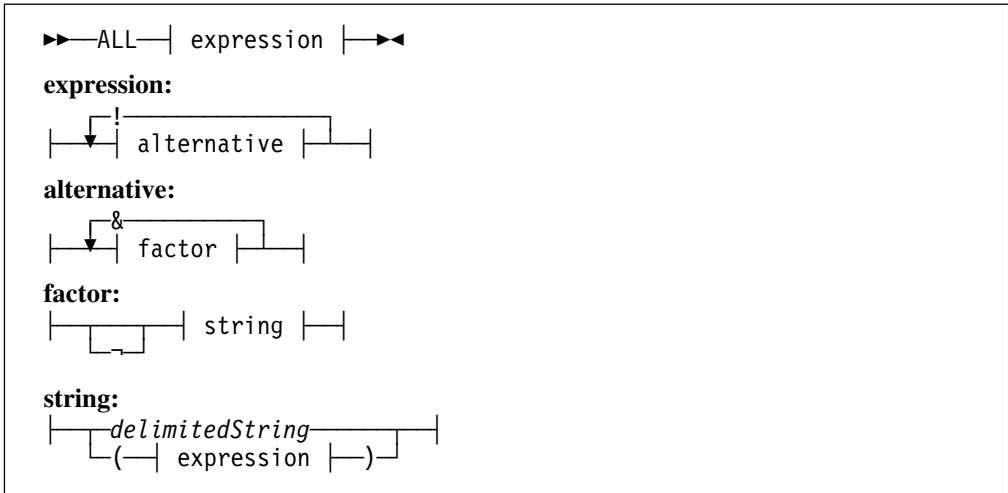
Notes:

1. *aggrc* can aggregate the return codes written to the secondary output stream by host command processors.
2. No output is produced if there is no input.

Return Codes: When the output record cannot be written (the stage is last in a pipeline), the return code is set to the aggregate of the numbers read.

all—Select Lines Containing Strings (or Not)

all selects records that satisfy a specified search criterion. In addition to XEDIT-like string expressions, *all* supports the use of parentheses to group expressions.



Type: Selection stage.

Syntax Description: The argument is a string expression. A string expression consists of alternatives delimited by exclamation marks or vertical bars (! or |). An alternative consists of terms delimited by ampersands (&). A term has an optional leading not symbol (¬) followed by a delimited string or an expression in parentheses. That is, the precedence of the operators is ¬ (highest), &, and ! (lowest).

Blanks are optional between operators and strings.

Operation: *all* constructs a suitable multistream subroutine pipeline containing *locate* and *nlocate* filters, and runs it. Records that match the criterion are written to the primary output stream; others are written to the secondary output stream when connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *all* strictly does not delay the record.

Commit Level: *all* starts on commit level -2. It verifies that the secondary input stream is not connected, parses the expression, and then issues a subroutine pipeline to work on data. This subroutine commits to level 0 in due course.

Premature Termination: *all* terminates when it discovers that no output stream is connected.

See Also: *locate* and *nlocate*.

Examples: To select records that contain a string or a vertical bar (4F in hexadecimal), or both:

```
all /abc/ ! x4f
```

To select records that contain an exclamation mark and either (or both) of two strings:

```
...| all (/abc/ ! /def/) & !/ |...
```

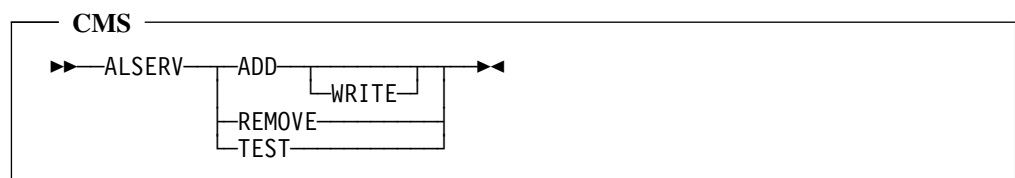
Note that XEDIT does not support parentheses for grouping.

Notes:

1. Performance will improve if the equivalent multistream pipeline network is specified instead of using *all*.
2. Specify a percent sign followed by a keyword (in lower case) at the beginning of the argument string to inspect the subroutine pipeline that is constructed. Specify %debug to write it to the file ALL DEBUG. Specify %dump to write the subroutine pipeline to the primary output stream before invoking it.

alserv—Manage the Virtual Machine’s Access List

alserv interfaces to the CP macro ALSERV.



Type: Host interface.

Syntax Description:

WRITE Give ALET write access to the data space.

Operation: *alserv* reads the primary input stream.

ADD Create an ALET in the access list. The ALET has write permission when WRITE is specified and the owner has granted your virtual machine write access.

REMOVE Remove an ALET from the access list.

TEST Verify the correctness of an ALET. Correct ALETs are passed to the primary output stream. Incorrect ones are passed to the secondary output stream, if it is connected; otherwise the stage terminates with an error message.

Input Record Format:

ADD Address space identification tokens (ASIT) in the first eight bytes. The record may be eight, twelve, or sixteen bytes long.

REMOVE Four bytes access list entry token (ALET).

TEST

Output Record Format:

ADD Access list entry tokens (ALET).

REMOVE The input record is passed.

TEST

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *alserv* does not delay the record.

Commit Level: *alserv* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *alserv* terminates when it discovers that no output stream is connected.

See Also: *adrspac*.

Examples: See Chapter 18, “Using VM Data Spaces with *CMS Pipelines*” on page 210.

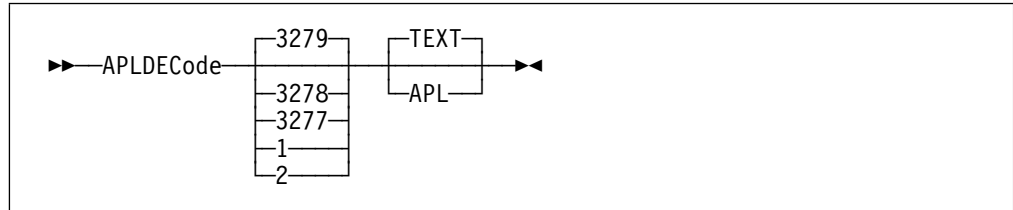
Notes:

1. The virtual machine must be in XC mode (this excludes z/CMS).

Publications: z/VM: CP Programming Services.

apldecode—Process Graphic Escape Sequences

apldecode processes input fields from a 3270 terminal with the APL/TEXT feature. This decodes two-byte escape sequences into single bytes.



Type: Arcane filter.

Syntax Description: Two keyword operands are optional. The first operand specifies the type of device for which data are decoded. The default is to decode with the X'08' graphic escape sequence used by devices such as 3278 and 3279 (and subsequent terminals). 3277 specifies the older style of decoding that uses pseudo start of field orders. The first operand can also be specified as 1 (for 3278) or 2 (for 3277); these correspond to the third word of the output from *fullscrs*. The second keyword specifies whether the TEXT (default) or APL mapping should be used.

Operation: Two translate tables are set up containing the defaults that correspond to the CP translation for TEXT ON (or APL ON if the keyword APL is specified) for the specified type of terminal. If the secondary input stream is defined, these defaults are modified by overlaying one record from it.

Input records on the primary input stream are scanned for escape characters, which are deleted. The character after an escape character is translated using the first table; other positions are translated using the second table. The escape character is X'08' when no operands are specified and when the first operand is 3279 (or 3278 or 1); the escape character is X'1D' when the first operand is 3277 (or 2).

Input Record Format: Inbound 3270 data without orders. In particular, SBA (set buffer address) orders should have been processed before the record is passed to *apldecode*.

If the secondary input stream is connected, a single record is read from it before the file on the primary stream is processed. (End-of-file is treated as if a null record were read.) This record can be any length, but only the first 512 bytes are used. The record is assumed to contain two translate tables that are to be overlaid on the two default translate tables, starting at the beginning of the first table. The ending part of the tables is left unchanged if the record is shorter than 512 bytes.

Streams Used: If the secondary input stream is defined, one record is read and consumed from it. The secondary input stream is severed before the primary input stream is processed. The secondary output stream must not be connected.

Record Delay: *apldecode* strictly does not delay the record.

Commit Level: *apldecode* starts on commit level -2. It verifies that the secondary output stream is not connected and then commits to level 0.

Converse Operation: *aplencode*.

See Also: *buildscr*.

Examples:

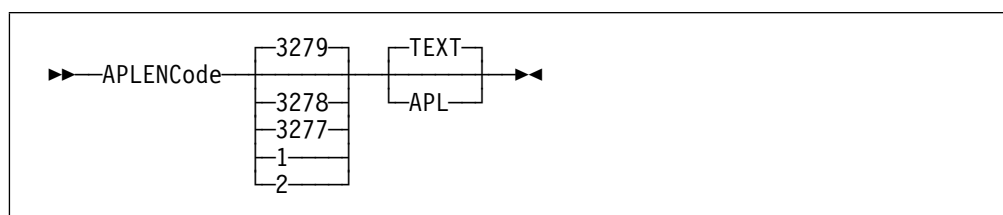
To decode the inbound stream from an unformatted 3278/9:

```
pipe literal {CEnter data: | fullscr | spec 4-* | apldecode | var input
```

The first stage of this pipeline generates the data to be displayed by *fullscr*. The first byte contains X'C0', which is a flag byte that specifies the erase/write alternate function. The second byte contains a write control character that restores the keyboard to allow user input (it also resets any modified data tags, which is not relevant in this example). The third stage discards the attention ID and the cursor address from the input record; because the screen is unformatted, the remaining data contain no 3270 device orders to worry about. The fourth stage decodes the graphic escape sequences and delivers a record that contains one byte per character to the final stage, which stores this into a variable.

aplencode—Generate Graphic Escape Sequences

aplencode encodes data to be displayed on a 3270 terminal with the APL/TEXT feature.



Type: Arcane filter.

Syntax Description: Two keyword operands are optional. The first operand specifies the type of device for which data are encoded. The default is to encode with the X'08' graphic escape sequence used by devices such as 3278 and 3279 (and subsequent terminals). 3277 specifies the older style of encoding that uses pseudo start of field orders. The first operand can also be specified as 1 (for 3278) or 2 (for 3277); these correspond to the third word of the output from *fullscr*. The second keyword specifies whether the TEXT (default) or APL mapping should be used.

Operation: Two translate tables are set up containing the defaults that correspond to the CP translation for TEXT ON (or APL ON if the keyword APL is specified) for the specified type of terminal. If the secondary input stream is defined, these defaults are modified by overlaying one record from it.

Characters in the input record for which the corresponding position in the first translate table is nonzero are replaced with an escape character and the value from the first translate table. Characters that are not to be escaped are translated according to the second translate table. The escape character is X'08' when no operands are specified and when the first operand is 3279 (or 3278 or 1); the escape character is X'1D' when the first operand is 3277 (or 2).

Input Record Format: Character data without 3270 orders.

If the secondary input stream is connected, a single record is read from it before the file on the primary stream is processed. (End-of-file is treated as if a null record were read.) This record can be any length, but only the first 512 bytes are used. The record is

append

assumed to contain two translate tables that are to be overlaid on the two default translate tables, starting at the beginning of the first table. The ending part of the tables is left unchanged if the record is shorter than 512 bytes.

Streams Used: If the secondary input stream is defined, one record is read and consumed from it. The secondary input stream is severed before the primary input stream is processed. The secondary output stream must not be connected.

Record Delay: *aplencode* strictly does not delay the record.

Commit Level: *aplencode* starts on commit level -2. It verifies that the secondary output stream is not connected and then commits to level 0.

Converse Operation: *apldecode*.

See Also: *buildscr*.

Examples: To build an unformatted screen containing a message to be displayed on a 3278:

```
pipe var message | aplencode | spec xc0c3 1 1-* next | fullscr noread
```

The message is encoded for a 3278 TEXT in the second stage. The third stage adds a flag byte (erase/write alternate) and a write control character (keyboard restore, reset modified data tags). The last stage displays the message on the terminal without waiting for operator action. Thus, the program that issued the pipeline continues immediately.

append—Put Output from a Device Driver after Data on the Primary Input Stream

append passes all input records to the output and then runs a device driver to generate additional output.

▶▶—APPEND—*string*—◀◀

Type: Control.

Syntax Description: The argument *string* is normally a single stage, but any pipeline specification that can be suffixed by a connector (*|*:*) is acceptable (see usage note 2).

Operation: All records on the primary input stream are copied to the primary output stream. Then the *string* is issued as a subroutine pipeline with CALLPIPE, using the default stage separator (*|*), double quotes as the escape character (*"*), and the backward slash as the end character (**). The beginning of the pipeline is unconnected. The end of the pipeline is connected to *append*'s primary output stream. (Do not write an explicit connector.)

In the subroutine pipeline, device drivers that reference REXX variables (*rexxvars*, *stem*, *var*, and *varload*) reach the EXEC COMM environments in effect for *append*.

Streams Used: *append* shorts the primary input stream to the primary output stream (*callpipe *:*|*:*); this does not delay the record. The specified *string* can refer to all defined streams except for the primary output stream (which is connected to the end of the subroutine pipeline by *append*); the primary input stream will be at end-of-file.

Record Delay: *append* strictly does not delay the record. The records that are appended are delayed until the end of the input file.

Premature Termination: *append* terminates if it is unable to copy all input to the output before it issues the subroutine pipeline to run the specified string.

See Also: *preface*.

Examples: To append the contents of a variable to the stream being built:

```
...| append var lastline |...
```

append is also used to append a line of literal data:

```
...| append literal *** End of data ***|...
```

Notes:

1. *append* is useful to add literals after a file has been reformatted, for instance with *spec*.
2. The argument string may contain stage separators and other special characters. Be sure that these are processed in the right place. The argument string is passed through the pipeline specification parser twice, first when the pipeline containing the *append* stage is set up, and secondly when the argument string is issued as a subroutine pipeline. The two example pipelines below show ways to append a subroutine pipeline consisting of more than one stage. In both cases, the *split* stage is part of the subroutine pipeline and, thus, splits only the record produced by the second literal stage:

```
pipe literal c d e| append literal a b || split | console
▶c d e
▶a
▶b
▶Ready;
```

```
pipe (sep ?) literal c d e? append literal a b | split ? console
▶c d e
▶a
▶b
▶Ready;
```

In the first example, the stage separator that is to be passed to the subroutine pipeline is self-escaped in the main pipeline. In the second example, the stage separator for the main pipeline is a question mark; thus, no special treatment is required to pass the stage separator (|) to *append*.

Now consider how to specify a vertical bar as part of an argument to the subroutine pipeline (in both cases, the variable data has the value `abc|def`):

```
pipe var data | append var data || split ||| | console
▶abc|def
▶abc
▶def
▶Ready;
```

```
pipe (stagesep ?) var data ? append var data | split || ? console
▶abc|def
▶abc
▶def
▶Ready;
```

In the first example, the stage separator that should be recognised in the subroutine pipeline is self-escaped; to get the parameter (a single |) through the pipeline specification parser twice, it must be doubly self-escaped; that is, the four vertical bars become one when the argument is presented to *split*. In the second example, the main pipeline uses the question mark as its stage separator and thus no escape is required to pass the vertical bar to the subroutine pipeline; and a single self-escape suffices to get the vertical bar to *split*.

3. Because a subroutine pipeline is used to pass the input to the output, it will terminate prematurely if the output is not connected or if end-of-file propagates backwards in the pipeline. Ensure that the output is connected when using a cascade of *hole* and *append* to issue a command after the input stream reaches end-of-file:

```
... | hole | append command ... | hole
```

Without the second *hole*, the command stage might start before you intended it to.

4. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: The return code is the return code from the CALLPIPE pipeline command. It may reflect errors in the argument string or trouble with the stage(s) in the pipeline.

asatomc—Convert ASA Carriage Control to CCW Operation Codes

asatomc converts ASA carriage control characters (in the first position of each record) to the corresponding machine carriage control characters; these are stored in the first position of the previous record to turn an immediate carriage movement into a delayed one. *asatomc* passes a file that already has machine carriage control characters unchanged, verifying that all records have correct machine carriage control characters.



Type: Filter.

Operation: Records that have X'03' (no operation) in column one at the beginning of the file are passed to the output unchanged. If the first input record has a valid machine carriage control character, the input is passed unmodified to the output and each record is verified to have a valid machine carriage control character.

Input Record Format: The first column of the record is an ASA carriage control character:

+	Overprint the previous record.
(blank)	Print on the next line.
0	Skip one line and print a line.
- (hyphen)	Skip two lines and print a line.
1-9	Skip to the specified channel and print a line.
A-C	Skip to channel 10 through 12 and print a line.

Output Record Format: The first column of the record is a machine carriage control character:

xxxx x001	Write the data part of the record and then perform the carriage operation specified by the five leftmost bits.
xxxx x011	Perform the carriage operation defined by the five leftmost bits immediately (the data part of the record is ignored).
000n n0x1	Space the number of lines (0 through 3) specified by bits 3 and 4.
1nnn n0x1	Skip to the channel specified by bits 1 through 4. The number must be in the range 1 to 12 inclusive.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: When the carriage control character is converted (as opposed to being passed through unmodified), the carriage control character is not delayed; the data part of a record is delayed to the following record.

Premature Termination: *asatomc* terminates when it discovers that its output stream is not connected.

Converse Operation: *mctoasa*.

Examples:

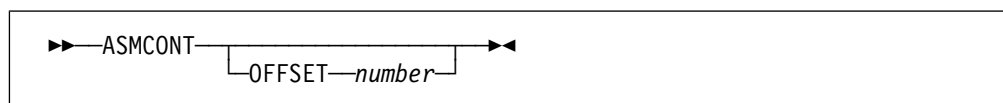
```
pipe literal 1Head line | asatomc | spec 1 c2x 1 2-* next | console
▶8B
▶01Head line
▶Ready;
```

Notes:

1. The last output record has a write no space command code (X'01').

asmcont—Join Multiline Assembler Statements

asmcont writes a line for each Assembler statement it reads. Continuation lines are joined to the first line of the statement.



Type: Filter.

Syntax Description:

OFFSET	The assembler statements are not at the beginning of the record. For example, a listing file often has the Assembler statement at offset 41.
number	Specify an offset that is zero or positive. The default offset is zero.

Operation: When *OFFSET* is specified, the column numbers in the following description should be increased by the number specified. The contents of the offset columns are deleted from continuation records, but kept on the first record of a statement.

A record shorter than 72 characters or with a blank character in column 72 is copied to the output.

asmfind

When column 72 of a record is a non-blank character, columns 1-71 are loaded into a buffer. Records are read up to one that is shorter than 72 characters or has a blank character in column 72. The contents of columns 16-71 (or the end of the record) are appended to the buffer; the contents of columns 1 through 15 are discarded; they are not inspected to verify that they are blank. The contents of the buffer are written when the end of the statement is reached.

Input Record Format: An Assembler statement consists of one or more lines. Lines before the last one have a non-blank character in column 72. The last line of a statement is blank in column 72, or shorter than 72 characters.

Record Delay: When column 72 is blank, *asmcont* strictly does not delay the record. Records that are not blank in column 72 are delayed until the next record that is blank in column 72.

Premature Termination: *asmcont* terminates when it discovers that its output stream is not connected.

Converse Operation: *asmxpnd*.

Examples: To find statements with the string 'R13':

```
! /* Find R13s */
! 'PIPE',
!   | < sample assemble',
!   | asmcont',
!   | locate /R13/',
!   | asmxpnd',
!   | console'
```

Notes:

1. *asmcont* does not support changes to the statement format by the ICTL Assembler instruction.

asmfind—Select Statements from an Assembler File as XEDIT Find

asmfind selects Assembler statements that begin with the specified string. It discards statements that do not begin with the specified string. An Assembler statement can span lines. XEDIT rules for FIND apply.



Type: Selection stage.

Syntax Description: A string is optional. The string starts after exactly one blank character. Leading and trailing blanks are significant. The maximum string length is 71 characters.

Operation: Input records are matched the same way XEDIT matches text in a FIND command (tabs 1, image off, case mixed respect):

- A null string matches any record.
- Blank characters in the string represent positions that must be present in the input record, but can have any value.

- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

When the first line of a statement is matched, *asmfind* copies all lines of the statement without further inspection to the primary output stream, or discards them if the primary output stream is not connected. When the first line of a statement is not matched, *asmfind* discards all lines of the statement without further inspection, or copies them to the secondary output stream if it is connected.

Input Record Format: An Assembler statement consists of one or more lines. Lines before the last one have a non-blank character in column 72. The last line of a statement is blank in column 72, or shorter than 72 characters.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *asmfind* strictly does not delay the record.

Commit Level: *asmfind* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *asmfind* terminates when it discovers that no output stream is connected.

Converse Operation: *asmnfind*.

See Also: *asmcont* and *asmxpnd*.

Examples: To select all statements in an Assembler program that have a label beginning with 'LAB':

```
... | asmfind LAB|...
```

To select all statements in an Assembler program that have the label 'LAB':

```
... | asmfind LAB_|...
```

The underscore indicates that column 4 must be blank; thus the label is three characters.

To select all comments in an Assembler program:

```
...| asmfind *|...
```

To select all statements of an Assembler program, except comments and those having a label:

```
...| asmfind _|...
```

Notes:

1. *asmfind* does not support changes to the statement format by the ICTL Assembler instruction.
2. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the

asmnfind

comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

asmnfind—Select Statements from an Assembler File as XEDIT NFind

asmnfind selects Assembler statements that do not begin with the specified string. It discards statements that begin with the specified string. An Assembler statement can span lines. XEDIT rules for NFind apply.



Type: Selection stage.

Syntax Description: A string is optional. The string starts after exactly one blank character. Leading and trailing blanks are significant. The maximum string length is 71 characters.

Operation: Input records are matched the same way XEDIT matches text in an NFind command (tabs 1, image off, case mixed respect):

- A null string matches any record.
- Blank characters in the string represent positions that must be present in the input record, but can have any value.
- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

When the first line of a statement is not matched, *asmnfind* copies all lines of the statement without further inspection to the primary output stream, or discards them if the primary output stream is not connected. When the first line of a statement is matched, *asmnfind* discards all lines of the statement without further inspection, or copies them to the secondary output stream if it is connected.

Input Record Format: An Assembler statement consists of one or more lines. Lines before the last one have a non-blank character in column 72. The last line of a statement is blank in column 72, or shorter than 72 characters.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *asmnfind* strictly does not delay the record.

Commit Level: *asmnfind* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *asmnfind* terminates when it discovers that no output stream is connected.

Converse Operation: *asmfind*.

See Also: *asmcont* and *asmxpnd*.

Examples: To select labelled or comment statements from an Assembler program:

```
...| asmnfind _|...
```

Notes:

1. *asmnfind* does not support changes to the statement format by the ICTL Assembler instruction.
2. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

asmxpnd—Expand Joined Assembler Statements

asmxpnd spans input records over 80-column “cards” using Assembler conventions for continuation lines.

►►—ASMXPND—◄◄

Type: Filter.

Operation: For each input record, one or more 80-byte records are created with continuation characters, as required, in column 72. Columns 1 through 71 of the input record are written as the first line of a statement, padded with blanks if required. The statement is not continued when the input record is 71 bytes or shorter: column 72 is made blank. When the input record is longer than 71 characters, continuation is indicated with an asterisk in column 72 of the first record for the statement. The remaining characters are written to continuation records with blanks in columns 1 through 15, 56 characters for each record. Continuation is indicated in each output record until the input record is exhausted.

Output Record Format: 80-byte records with blanks in column 73-80.

Record Delay: *asmxpnd* does not delay the last record written for an input record.

Premature Termination: *asmxpnd* terminates when it discovers that its output stream is not connected.

Converse Operation: *asmcont*.

Examples: To generate an Assembler DC instruction for each input line with the contents of the line as a character constant:

```
/* MAKEDC REXX */
'callpipe (name MAKEDC)',
  '|*:',
  '|change /'/'/'/', /* double quotes */
  '|change /&/&/', /* ... and ampersands */
  '|spec /DC/ 10 /C'/ 16 1-* next /'/' next', /* Make big DC */
  '| / / next', /* ensure S&D area available */
  '|asmxpnd', /* Continuation if needed */
  '|*:'
exit RC
```

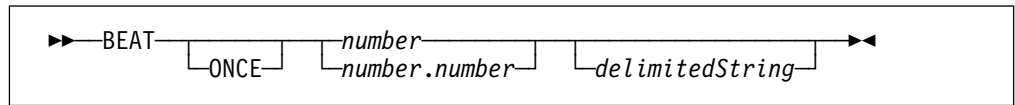
beat

Notes:

1. *asmxpd* does not support changes to the statement format by the ICTL Assembler instruction.

***beat*—Mark when Records Do not Arrive within Interval**

beat is used as a “heartbeat monitor”. It passes records from its primary input stream to its primary output stream. If no input record arrives within the specified interval, a record is written to the secondary output stream. A record can be written to the secondary output stream once per input record or each time the interval expires.



Type: Device driver.

Syntax Description: An initial keyword is optional. One numeric operand is required. A delimited string is optional.

The numeric operand specifies the interval in seconds. Up to six digits may be specified after the period, allowing for a microsecond interval.

The default string is a null string.

Operation: The delimited string is written to the secondary output stream when the interval expires without an input record arriving. When `ONCE` is omitted, *beat* restarts another timeout immediately after the output record on the secondary output stream is consumed; when `ONCE` is specified, *beat* waits for the next input record without generating further output records.

Streams Used: Two streams must be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *beat* strictly does not delay the record it passes to the primary output stream.

Commit Level: *beat* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *beat* terminates when it discovers that either of its output streams is not connected.

See Also: *delay*, *gate*, and *synchronise*.

Examples: To pass records until no record has arrived in one second:

```
'pipe (end ?) ... | b: beat 1 | ... ? b:'
```

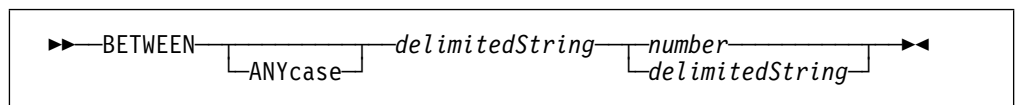
In this example, the secondary output stream is defined, but it is not connected. This causes *beat* to terminate when it tries to write a null record to its secondary output stream.

Notes:

1. For *beat* to work as described, it is assumed that the input records are read into the pipeline by a device driver that obtains its input through an asynchronous host interface, such as does *tcpclient*.
2. *beat* cannot detect timeouts during CMS commands.

between—Select Records Between Labels

between selects groups of records whose first record begins with a specified string. The end of each group can be specified by a count of records to select, or as a string that must be at the beginning of the last record.



Type: Selection stage.

Syntax Description: A keyword is optional. Two arguments are required. The first one is a delimited string. The second argument is a number or a delimited string. The number must be 2 or larger.

Operation: *between* copies the groups of records that are selected to the primary output stream, or discards them if the primary output stream is not connected. Each group begins with a record that matches the first specified string. When the second argument is a number, the group has as many records as specified (or it extends to end-of-file). When the second argument is a string, the group ends with the next record that matches the second specified string (or at end-of-file).

When ANYCASE is specified, *between* compares fields without regard to case. By default, case is respected.

between discards records before, between, and after the selected groups or copies them to the secondary output stream if it is connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *between* strictly does not delay the record.

Commit Level: *between* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *between* terminates when it discovers that no output stream is connected.

Converse Operation: *outside*.

See Also: *inside*, *notinside*, and *pick*.

Examples: To select examples in a Script file, assuming the tags are at the beginning of the record:

```
... | between /:xmp./ /:exmp./ |...
```

block

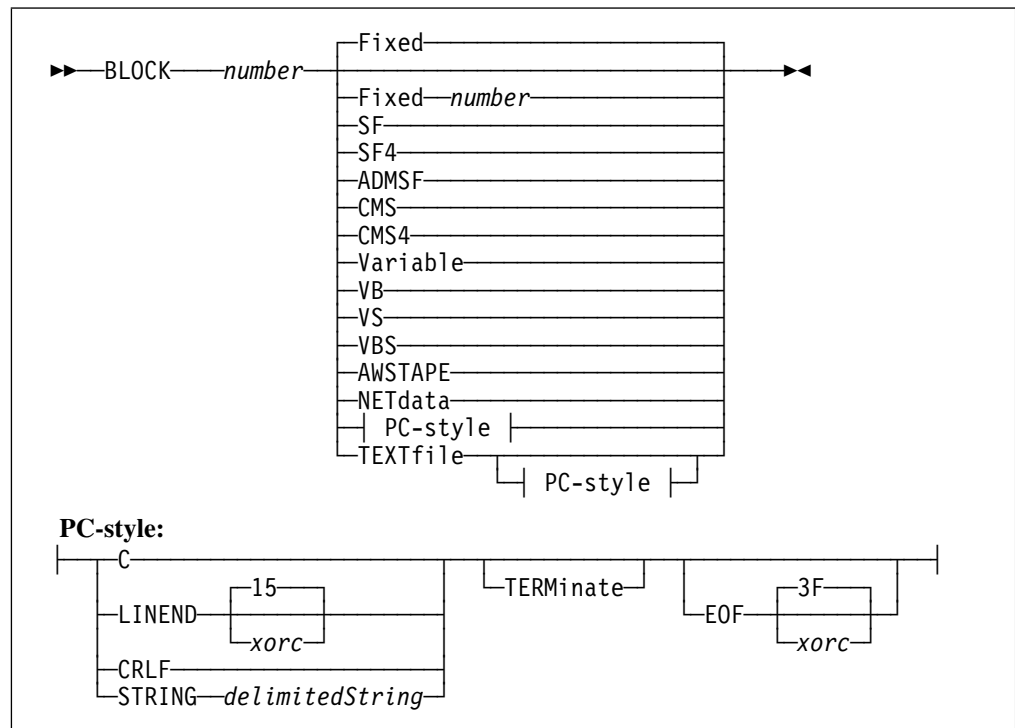
Notes:

1. *between* selects records from multiple groups, whereas *frlabel* followed by *tolabel* selects only one group.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
3. *pick* can do what *between* does and much more.

***block*—Block to an External Format**

block generates output *blocks* from input *logical records*. The output blocks are in a format that is suitable for interchange with other systems. For instance, a tape in the VBS format can be read on an z/OS system; a file blocked C and translated to ASCII can be downloaded to a personal computer as a binary object.

Some blocking formats fill all output blocks (possibly except for the last) to capacity. As a consequence data from an input record may be *spanned* over two or more output blocks.



Type: Filter.

Syntax Description: The first word of the argument string must be a number specifying the maximum block size. The blocking format is an optional keyword; the default is FIXED. The record length may be specified after the keyword FIXED. A line end character is optional with LINEND; the default is 15 (representing the character X'15').

The minimum acceptable value for the block size is 1, except for ADMSF, where it is 2; except for AWSTAPE, where it is 7; and except for V, VB, VS, and VBS, where it is 9. There is no limit to the block size other than the availability of virtual storage. Even though no maximum block size is enforced for AWSTAPE, a maximum of 4096 should be observed; files created with a block size larger than 4096 may not function with the real AWSTAPE device driver.

The formats C, LINEND, CRLF, and STRING support two additional keywords, TERMINATE and EOF. Use TERMINATE to specify that the last line of the file should have a terminating line end sequence; the default is to insert line end sequences between lines and leave the last line without one. Use EOF to specify that an end-of-file character should be appended to the contents of the file. X'3F' (substitute) is the default end-of-file character. *block* inserts an end-of-file character only when the keyword is specified.

The format TEXTFILE supports two additional options: A line end character specified as C, LINEND, CRLF, or STRING; and the EOF option. The TERMINATE option can be specified, but it cannot be suppressed. The default is LINEND 15 TERMINATE.

Operation: Input records are blocked or spanned to blocks of the specified size or less. Descriptor words are generated for formats other than FIXED, C, CRLF, STRING, and LINEND.

FIXED	Juxtapose records with no control information in between. Only the last output block can be short. Null input records are discarded. When a record length is specified (as the second number), it is ensured that all input records that are not null are of this length. When a record length is not specified, the length of the first record that is not null is used as the record length; it is ensured that all input records have the same length and that the block size is a multiple of the record length. <i>block</i> stops with an error message if this is not the case. The corresponding z/OS record format is Fixed Block Standard (RECFM=FBS).
SF	Block records to the structured field format. Each record is prefixed by a halfword length field; the length includes the length of the halfword (thus, the minimum length is 2). Except for possibly the last block, each block is filled completely; a logical record (and the halfword length field) can span blocks.
SF4	Block records to a format similar to the structured field format. Each record is prefixed by a fullword length field; the length includes the length of the fullword (thus, the minimum length is 4). Except for possibly the last block, each block is filled completely; a logical record (and the fullword length field) can span blocks.
ADMSF	Block records to the structured field format used in GDDM objects. Each record is prefixed by a halfword length field; the length includes the length of the halfword (thus, the minimum length is 2). Except for possibly the last block, each block is filled completely; a logical record can span blocks. The length field does not span blocks. When the last byte of a logical record is stored in the second last position of the output record, the last position is padded with X'00'; the following record is stored at the beginning of the next block.
CMS	Build blocks in the format used internally by the CMS file system for files with variable length records. This format is exposed in data unloaded by, for instance, the CMS commands TAPE DUMP and DISK DUMP. The record descriptor word prefixed to the record is a halfword (two bytes). It contains the length of the record (not including the halfword). Logical records (and record descriptors) can span output blocks. The last block is not padded. (The TAPE DUMP command does not pad blocks; use <i>pad</i> to pad the last block with zeros as it is in the file system.) Null input records are discarded. <i>block</i> CMS terminates with an error message if an input record is 64K or longer.

block

CMS4	Prefix each record with a fullword length field which contains the count of characters that follow the length field. That is, the length of the length field is not included in the count. Null input records are discarded. Logical records (and record descriptors) can span output blocks. The last block is not padded.
VARIABLE	Block to the OS unblocked variable format (RECFM=V). Each record is prefixed with a block descriptor word and a record descriptor word, increasing its length by eight bytes. <i>block</i> V terminates with a message if a record is longer than 32759 bytes or is too long to fit in the output buffer (that is, if the length of the input record is greater than the specified block size minus eight).
VB	Block to the OS variable blocked format (RECFM=VB). The output block contains as many logical records as will fit within the specified block size. The overhead for descriptor words is four bytes per block plus four bytes per record. A logical record does not span output blocks. <i>block</i> VB terminates with a message if a record is longer than 32759 bytes or is too long to fit in the output buffer (that is, if the length of the input record is greater than the specified block size minus eight).
VS	Generate blocks in the OS variable spanned format (RECFM=VS). Records are segmented to fit within the output buffer. A block has one segment. A logical record starts a new block.
VBS	Block to the OS variable blocked spanned format (RECFM=VBS). Records are segmented to fit within the output buffer. A logical record can span blocks; there is a segment in each block that the record is spanned over. Segment descriptors are prefixed to the segments; segment descriptors do not span blocks. The output block length (except possibly for the last block) is between n-5 and n, where n is the specified block size.
AWSTAPE	Segment and span records according to the format used by the AWSTAPE device driver for p370, rs370, p390 (PC Server System/390) and zPDT (IBM System z Personal Development Tool). If the input record plus the six bytes segment descriptor is not longer than the specified block size, a single output record is produced. For longer input records, as many segments as required are produced.
NETDATA	Segment and span records according to the NETDATA format. Null input records are discarded. The first byte of each input record that is not null is a flag byte indicating whether the record is a control record (the bit for X'20' is on) or a data record (the bit for X'20' is off). Bits 3 to 7 of the flag byte are copied to the corresponding bits of the flag bytes in all output segments produced from an input record. The NETDATA control records must have been built previously, most likely injected by one or more <i>literal</i> or <i>preface</i> stages. The output buffer is flushed after an \NMR6 record has been written. This ensures that any stacked file begins in a separate record.
C	Block records with an end of line character (line feed, X'25') between logical records. Line feed control characters in input records are copied unchanged to the output. Except for possibly the last block, each block is filled completely; a logical record can span blocks.

LINEND	Block records with an end of line character (by default new line, X'15') between logical records. End of line characters in input records are copied unchanged to the output. Except for possibly the last block, each block is filled completely; a logical record can span blocks.
CRLF	Block records with carriage return and line feed (X'0D25') between logical records. The values are EBCDIC; blocking should be done before the records are translated to ASCII. Except for possibly the last block, each block is filled completely; a logical record can span blocks, as can a line end sequence.
STRING	Block records with the specified string between logical records. Except for possibly the last block, each block is filled completely; a logical record can span blocks, as can a delimiter string.
TEXTFILE	Append a line end sequence to each record and join as many records as will fit in the buffer. Records are not spanned across block boundaries. If the record and the line end sequence cannot fit in the buffer, the record and the line end sequence are written as separate records. The default line end character is X'15'. If EOF is specified, the end-of-file character is appended to the last record (after the line end sequence). TEXTFILE is designed for use with a byte stream file system.

Output Record Format: When V, VB, VS, or VBS is specified, the output block is prefixed by four bytes block descriptor. In a basic block descriptor, the length of the block (including the block descriptor word) is stored in the first two bytes of the block descriptor word; the first bit and the last two bytes are zero. An extended block descriptor word is used when the output block is longer than 32K. Its leftmost bit is one to distinguish the extended block descriptor from the basic format block descriptor; the length of the block including the block descriptor is stored in the following 31 bits.

When SF, SF4, ADMSF, V, VB, CMS, or CMS4 is specified, a record descriptor word is prefixed to each logical record. (That is, to each input record.) For V and VB, the record descriptor word is four bytes. The length of the record (including the length of the record descriptor word) is stored in the first two bytes; the next two bytes contain zeros. For SF and ADMSF, the record descriptor word is two bytes; it contains the length of the record (including the record descriptor). For SF4, the record descriptor word is four bytes; it contains the length of the record (including the record descriptor). For CMS, the record descriptor word is two bytes; it contains the length of the record (excluding the record descriptor). For CMS4, the record descriptor word is four bytes; it contains the length of the record (excluding the record descriptor).

When VS, VBS, AWSTAPE, or NETDATA is specified, a segment descriptor word is prefixed to each segment of a record. For VS and VBS, the the segment descriptor word is four bytes; the length of the segment (including the segment descriptor word) is stored in the first two bytes of the segment descriptor word; the third byte has segmentation flags (X'02' means not first segment, X'01' means not last segment); the last byte is zero. VS and VBS segments do not span blocks. For AWSTAPE, the segment descriptor is six bytes, consisting of two halfword length fields which have the least significant byte leftmost, a flag byte, and a byte of zeros. The first length field contains the number of data bytes that follow the segment descriptor; the second length field contains the number of data bytes in the previous segment (thus allowing for read backwards). In the flag byte, X'80' means the first segment of a physical block; X'40' means an end-of-file record; and X'20' means the last segment of a physical block. For NETDATA, the segment descriptor is a byte with the length of the segment (including the descriptor) followed by a flag byte

block

(X'80' means first segment; X'40' means last segment; X'20' means the record is a control record); segments can span blocks.

Streams Used: Records are read from the primary input stream and written to the primary output stream.

Record Delay: *block* delays input records as required to build an output record. The delay is unspecified.

Commit Level: *block* starts on commit level -2000000000. It allocates the buffer of the specified size and then commits to level 0.

Premature Termination: *block* terminates when it discovers that its output stream is not connected.

Converse Operation: *deblock*.

See Also: *fblock* and *join*.

Examples: To write a CMS file to an unlabelled tape in variable blocked format suitable for z/OS:

```
pipe < input file | block 16000 vb | tape
```

To write a fixed record format CMS file that has record length 80 in a format suitable for z/OS:

```
pipe < input file | block 16000 | tape
```

These examples show the effect of the TERMINATE and EOF options:

```
pipe literal abc|block 80 linend * | console
```

```
▶abc
▶Ready;
```

```
pipe literal abc|block 80 linend * terminate | console
```

```
▶abc*
▶Ready;
```

```
pipe literal abc|block 80 linend * terminate eof + | console
```

```
▶abc*+
▶Ready;
```

To turn a tape reel into an file that can be downloaded:

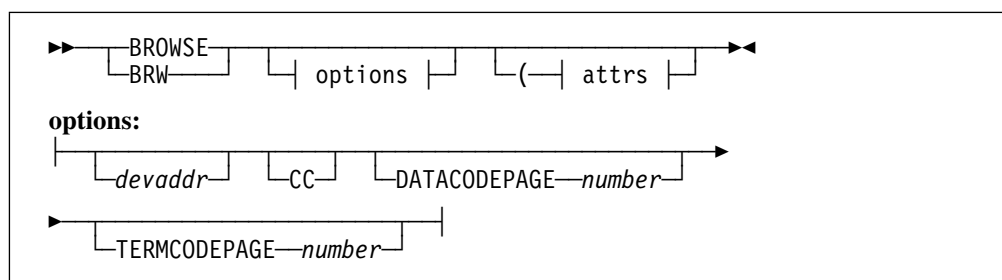
```
/* CMS2AWS REXX -- Build downloadable tape. */
'addpipe *.output:|block 4096 awstape |> maint 181 a'
Do until records=0 /* Two TMs are EOT */
  'callpipe (end ? name CMS2AWS)',
    '?tape', /* Read real tape */
    '|c: count lines', /* count number of blocks */
    '|*:', /* Pass a file on */
    '?c:',
    '|var records' /* Number of blocks in physical file */
  'output' /* Null record for tape mark */
end
```

Notes:

1. Refer to “Netdata Format” on page 65 for usage information about *block* NETDATA. The file INMR123 REXX is shipped on the CMS system disk (190).
The Netdata format is documented in *z/VM: CMS Macros and Functions Reference*, SC24-6262.
2. *block* C is a convenience for *block* LINEND 25. *block* CRLF is a convenience for *block* STRING X0D25.
3. LINEEND is a synonym for LINEND.
4. Though input records are called logical records and output records are called blocks, these are still records (or lines) as perceived by the pipeline dispatcher.

***browse*—Display Data on a 3270 Terminal**

browse displays its input data on a 3270 terminal. You can control the display with the program function keys. The display can be your terminal or (on CMS only) a terminal that is dialled to your virtual machine. The input is passed to the output as it is being displayed.



Type: Experimental device driver.

Syntax Description: The arguments consist of options, a left parenthesis, and attribute characters as defined for *buildscr*. The options specify the terminal to use (the log on terminal is the default), a keyword to specify that the data to be displayed contain machine carriage control characters, and keywords to specify the code pages of the data to display and of the terminal.

A left parenthesis separates the options for *browse* from an option list passed as the arguments to *buildscr*. In this option list, the first four words may each be specified as an asterisk, three characters, or six hexadecimal characters. The characteristic of the terminal is provided for any remaining unspecified options; an asterisk can be used as a placeholder when an option is specified and an earlier option is to be defaulted.

Operation: The geometry and features of the device are determined. The input file is shown in panels.

When CC is specified, the first column of the input record contains a carriage control character which may be an ASA or machine carriage control character. Each page begins with a page eject carriage control; a page is displayed in as many panels (the size of the display) as are required. When CC is omitted, input lines are displayed single spaced; a page is the size of the display.

Program Function Keys:

browse

- 1 Pop up a panel showing a summary of the program function key actions.
- 2 Unused; ignored.
- 3 Exit.
- 4 Move to the beginning of the file.
- 5 Move to the end of the file.
- 6 Redo the previous search.
- 7 Move a panel back (towards the beginning of the file).
- 8 Move a panel forward (towards the end of the file).
- 9 Unused; ignored.
- 10 Move a page back (towards the beginning of the file).
- 11 Move a page forward (towards the end of the file).
- 12 Exit.

Program function keys 13 through 24 perform the same function as keys 1 through 12, respectively.

The enter key moves a panel forward.

On CMS, program access key 2 drops into the INTM terminal monitor program.

Searching: The pop up panel displayed in response to program function key 1 contains an input area. A search is performed when characters are entered in this input area (case and blanks are respected in this string). The search is towards the end of the file for the particular string entered in the case entered. When the specified string is found in the 3270 data streams that represent the panels, the corresponding panel is displayed; there is no indication of where on the panel the matching string was found. The string can consist partially or entirely of 3270 control sequences. The last panel of the file is shown when the search fails; there is no audible indication.

Streams Used: Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *browse* strictly does not delay the record.

Premature Termination: *browse* terminates when it runs out of storage. This is likely to be accompanied by several REXX error messages.

Examples: To display a file:

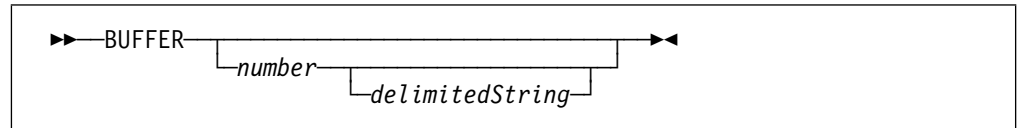
```
pipe < 5785rac memo | browse cc
```

Notes:

1. The part of the input that has been read by *browse* is kept in virtual storage; the file is read only as required to build panels for display or search. Thus, you can see the first few panels of even an infinitely large file.
2. When the output stream is connected, any unprocessed input records are passed to the output when you exit from *browse*.
3. *browse* does not support horizontal scrolling.

buffer—Buffer Records

buffer reads input records and accumulates them in memory. *buffer* writes the buffered records to its output when it reaches end-of-file and (if arguments are specified) each time it reads a null record. Optionally, *buffer* writes multiple copies of the buffered files.



Type: Filter.

Syntax Description: Specify a number to write multiple copies of a buffered file; a delimited string after the number specifies a record to be written between multiple copies of a buffered file.

Operation: When no arguments are specified, records (including null ones) are stored in a buffer until end-of-file. The buffered records are then written to the primary output stream in the order they were read.

When a number is specified (it can be 1), the input stream is considered to consist of one or more files separated by null records. Each file is stored in the buffer and written to the output (in the order it was read) when a null record is read, or at end-of-file. The set of records in a file is written as many times as specified by the first argument. If the number is 2 or more, the copies of a file are delimited by records containing the string specified by the second argument, or by null records if no second argument is specified. When the file has been written to the output as many times as requested, the null input record is copied to the output; the buffer is reset to be empty; and reading continues.

Streams Used: Records are read from the primary input stream and written to the primary output stream.

Record Delay: When no argument is specified, *buffer* delays all records until end-of-file. When an argument is specified, the null record (which terminates the part of the file being buffered) is consumed before writing the first line of the partial file.

Premature Termination: *buffer* terminates when it discovers that its output stream is not connected.

See Also: *copy*, *dup*, *elastic*, *instore*, *outstore*, and *sort*.

Examples: To read lines from the terminal and put them into the stack after the user signals end-of-file with a null input line:

```
/* Read user's input into stack */
say 'Enter commands for' gizmo:'
'PIPE console | buffer | stack'
```

If there were no *buffer* stage, the pipeline would loop as soon as a line was put into the stack, because the *console* stage would immediately read that line and write it to the *stack* stage, which would read it and put it back into the stack, and then the *console* stage would read it again.

With the *buffer* stage, the pipeline works as follows: *console* reads lines from CMS and writes them into the pipeline. It reads all lines from the console stack and the terminal

buffer

input queue before it begins reading from the terminal. It terminates when it reads a null line (which indicates end-of-file). *buffer* stores all its input lines in its buffer until it receives end-of-file on its input; it then writes the contents of its buffer to the output; and *stack* copies the records from its input to the CMS console stack. Thus, by the time *buffer* begins to write records, the *console* stage has terminated and it is safe to put the records onto the stack.

A direct read (see *console*) may be more appropriate for reading from the terminal to the stack.

A *buffer* stage is required to buffer the output from *rexxvars* when data derived from its output are stored back into the variable pool with a *var*, *stem*, *varload*, or *varset* stage:

```
pipe rexxvars | find v_ARRAY. | spec 3-* | buffer | stem vars.
```

As shown in this example, it may be more efficient to buffer the variables that are set rather than the output from *rexxvars*.

Notes:

1. A *buffer* stage may be needed to prevent stalls in intersecting multistream pipelines.
2. *dup* makes one or more copies of each input record; the copies are contiguous. *buffer* with a number makes copies of complete files.
3. With the same input, these two invocations of *buffer* produce identical output:

```
...| buffer   |...  
...| buffer 1 |...
```

But the timing of the output records is different if there are null records. When used without arguments, *buffer* reads all input records before it generates any output. When an argument is specified, *buffer* produces its first output record as soon as it reads the first null input record.

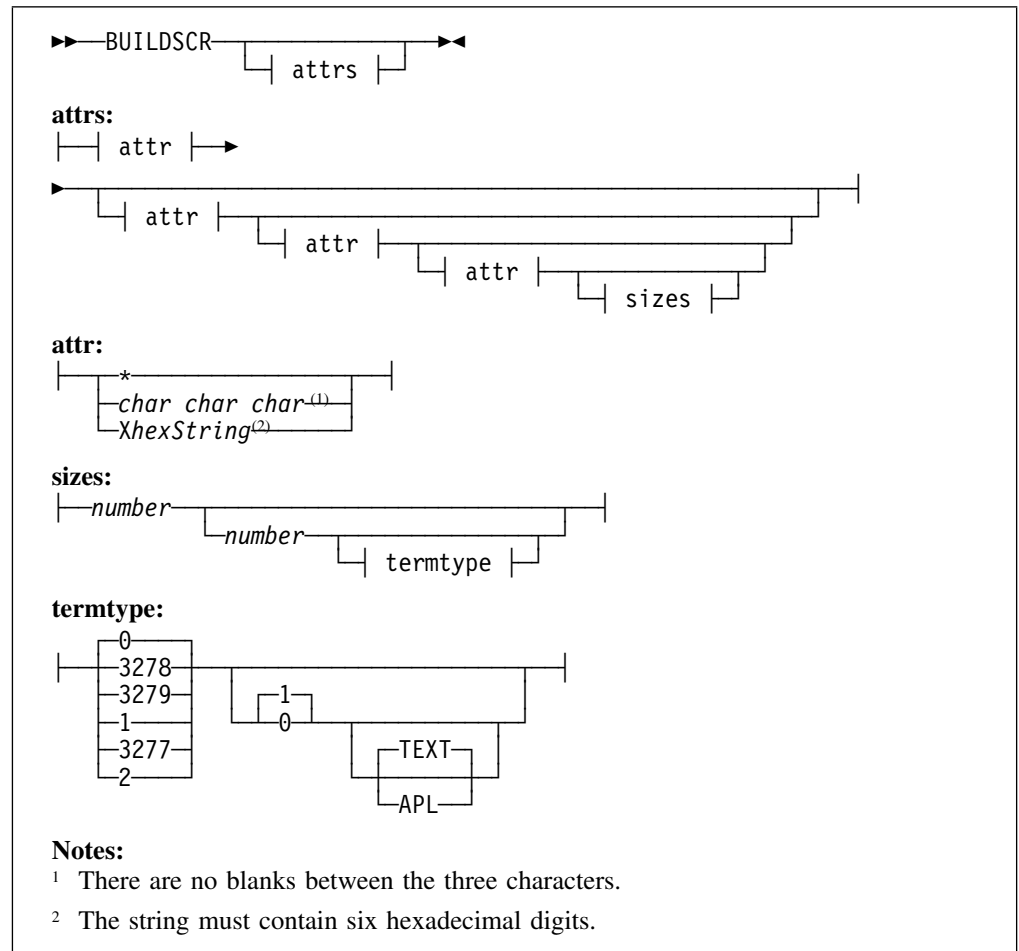
4. Use *buffer* without arguments to be sure that an arbitrary sequence of records is buffered in its entirety.
5. To generate two copies of an entire file that contains null lines:

```
...| instore | dup | outstore |...
```

The *dup* stage duplicates the single file descriptor record created by *instore*; *outstore* makes two copies of the file as a result.

buildscr—Build a 3270 Data Stream

buildscr prepares lines of data to be displayed on a 3270 display in full screen mode. The input to *buildscr* is a print file that has machine carriage control characters. Such a file is often generated by a preceding *overstr* stage. The output from *buildscr* contains 3270 character attributes that cause the highlighted and underscored data to be differentiated from normal text when the records are written to a 3270 (for instance by a *fullscr* stage).



Type: Arcane filter.

Syntax Description: The first four blank-delimited words specify extended attributes to be used for the four possible combinations of underscoring and highlighting. The order of the attribute items is: neither highlighted nor underscored, underscored, highlighted, and both highlighted and underscored. An attribute item may be an asterisk (*) to take the default or three characters that specify extended attributes. The three characters can be specified as such or as an X followed by six hexadecimal digits. The three attribute characters are in the order highlighting, colour, and program symbols. For instance, '27e' selects the programmed symbol set that has ID 'e' (=X'85') and makes it white (7) reverse video (2). Refer to *3274 Description and Programmer's Guide*, GA23-0061, for a description of 3270 extended attribute characters. Use X'00' to select the default attribute value depending on the terminal.

Words five and six contain numbers that specify the screen size in character cells (lines followed by columns). The default is 32 lines of 80 columns. The minimum size is 1920 (24 by 80); the maximum size is 16K.

The seventh word is a switch to indicate whether the display supports the APL/TEXT feature; it is assumed not to have the feature when the argument string contains six words or fewer. Specify 0 when the device does not support APL/TEXT. Specify 3278 (or 3279) for modern 3270 terminals; specify 3277 for the original 3277 terminals and some TELNET servers. You can also specify this operand in the form used for the third word of the output from *fullscrs*: The number 1 specifies 3278-style APL/TEXT; the number 2 specifies 3277-style APL/TEXT.

The eighth word specifies whether the terminal supports extended highlighting and character attributes; the default is 1. Specify 0 for a device that does not support extended features (3277 and some TELNET servers).

The ninth word specifies the type of APL/TEXT you wish to enable. The default is TEXT; specify APL to use such a mapping.

Operation: Two translate tables are set up containing the defaults that correspond to the CP translation for TEXT ON (or APL ON if the keyword APL is specified) for the specified type of terminal. If the secondary input stream is defined, these defaults are modified by overlaying one record from it. Output records are 3270 data streams that can be used to display the contents of the input file, formatted as on the page. Character attribute sequences are inserted into the data to switch attributes as determined by the contents of the corresponding positions of the descriptor record. An input page that has more lines than the display is written as several output records. Input lines that are too long for the screen width are truncated at the right hand side. Blank lines are generated for input lines that have skip carriage control (carriage control characters X'11', X'19', X'1B', X'0B', and X'13'). Skips to channels other than channel 1 are treated as requests to skip one line.

Input Record Format: X'00' in the first column indicates a descriptor record in the format produced by *overstr*. Each column of the descriptor record specifies the highlighting and underscoring of the corresponding column in the data record that follows the descriptor record. These descriptor values are used:

- X'00' The position is blank.
- X'01' The position contains an underscore. (An underscored blank.)
- X'02' The position contains a character that is neither blank nor underscore.
- X'03' The position contains an underscored character.
- X'04' The position contains a highlighted blank.
- X'05' The position contains a highlighted underscore.
- X'06' The position contains a highlighted (overprinted) character.
- X'07' The position contains a highlighted and underscored character.

Records without X'00' in column 1 must begin with a machine carriage control character; data are from column 2 onward. Records that are not preceded by a descriptor record are neither underscored nor highlighted (though they can contain underscore characters). The end of a page is indicated by a skip to channel 1 (X'89' or X'8B'). Data in a record that has X'89' carriage control are on the last line of a page.

The data part of input lines is truncated at the screen width.

If the secondary input stream is connected, a single record is read from it before the file on the primary stream is processed. (End-of-file is treated as if a null record were read.) This record can be any length, but only the first 512 bytes are used. The record is assumed to contain two translate tables that are to be overlaid on the two default translate tables, starting at the beginning of the first table. The ending part of the tables is left unchanged if the record is shorter than 512 bytes.

Output Record Format: The first position is a flag byte to indicate whether the screen image is the first from a page of the document (X'01') or a subsequent one (X'00'). Column 2 has the constant 'B' (X'C2'), which is the write control character for keyboard restore; 3270 orders and data follow. The screen is formatted to a single protected field with the attribute character in the lower right hand corner of the screen. The cursor is inserted on this attribute byte.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded. If the secondary input stream is defined, one record is read and consumed from it. The secondary input stream is severed before the primary input stream is processed. The secondary output stream must not be connected.

Record Delay: *buildscr* delays records until it has filled a panel. It is unspecified whether it reads one more record before writing the output record.

Premature Termination: *buildscr* terminates when it discovers that its output stream is not connected.

See Also: *fullscr*, *fullscrq*, *fullscrs*, *overstr*, and *xpndhi*.

Examples: To reformat Script output for a 1403 to be displayed on a 32-line 3270 that supports APL/TEXT:

```
pipe < $doc script | overstr | buildscr * * * * 32 80 1 |...
```

Refer to PIPDSCR EXEC (on MAINT 193) for a more sophisticated example.

Notes:

1. *buildscr* is intended to process the output from *overstr* (possibly with an intervening *xpndhi* stage).
2. Use *spec* to prefix X'09' (write no space) to each line of a file without carriage control.

```
...| spec x09 1 1-* 2 | buildscr |...
```
3. Use *asatmc* to convert from ASA carriage control to machine carriage control.
4. Use *fullscrs*, *fullscrq*, or diagnose 8C to determine the size of the terminal screen.
5. When specifying an attribute item in the operands for *buildscr*, use X'00' to select the default behaviour for a particular attribute. This is most conveniently done by specifying a hexadecimal string. Remember that all three characters of the word must be specified in hexadecimal; for example, the character “4” is specified as F4.

```
...| buildscr * xf40000 x00f700 xf4f700 |...
```

In this example, characters that are neither highlighted nor underscored are shown with the default highlighting, colour, and programmed symbol set. Underscored characters are shown using extended highlighting for underscore with the default colour and symbol set. Highlighted characters are shown in white.

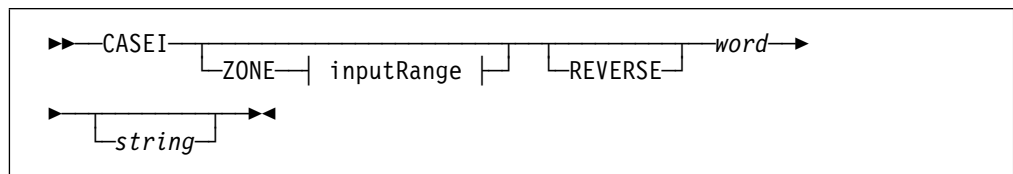
casei

6. The output from *buildscr* cannot in general be fed directly to *fullscr* to be displayed. A control mechanism is required for an interactive display, for instance to browse the data (see SCRCTL REXX on MAINT 193). At the least, the flag byte must be replaced by a control byte, as described for *fullscr*.
7. *buildscr* cannot ensure that the 3270 data stream is displayed on a device that has the geometry specified or defaulted. Lines may wrap when the actual screen has a different line length. The display may give unit check if the actual screen is smaller than the size specified (the product of lines and columns).

casei—Run Selection Stage in Case Insensitive Manner

The argument to *casei* is a stage to run. *casei* invokes the specified stage having translated the argument string to upper case. It then passes input records translated to upper case to this stage. When the stage writes a record to its primary output stream, the corresponding original input record is written to the primary output stream from *casei*; likewise, the original input record is written to the secondary output stream, when the stage writes to its secondary output stream.

The argument is assumed to be a selection stage; that is, it should specify a program that reads only from its primary input stream and passes these records unmodified to its primary output stream or its secondary output stream without delaying them.



Type: Control.

Syntax Description: A word (the name of the program to run) is required; further arguments are optional as far as *casei* is concerned, but the specified program may require arguments.

Operation: *casei* constructs a subroutine pipeline to perform the required transformation on input records. If ZONE is specified, only the specified part of the input record is passed to the specified stage (see *zone*). If REVERSE is specified, the contents of the input record are reversed (after the zone is selected).

Streams Used: Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *casei* does not add delay.

Commit Level: *casei* starts on commit level -2. It does not perform an explicit commit; the specified program must do so.

See Also: *reverse* and *zone*.

Examples: To select records that contain a particular GML tag at the beginning of the record, ignoring case:

```
pipe ... | casei find :figref | ...
```

Notes:

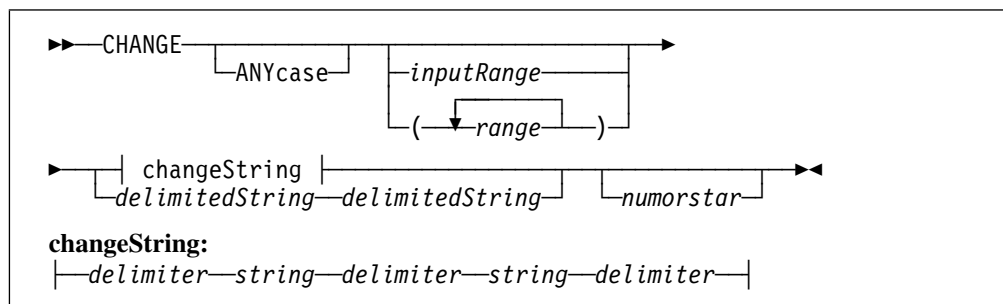
1. All built-in selection stages that operate on strings now support the ANYCASE option, as of *CMS Pipelines* level 1.1.10/0015; thus *casei* is obsolete as far as built-in programs are concerned. For example, a case insensitive *strfind* is:

```
pipe ... | strfind anycase /:figref / | ...
```
2. The argument string to *casei* is passed through the pipeline specification parser only once (when the scanner processes the *casei* stage), unlike the argument strings for *append* and *preface*.
3. End-of-file is propagated from the streams of *casei* to the corresponding stream of the specified selection stage.

Return Codes: If *casei* finds no errors, the return code is the one received from the selection stage.

change—Substitute Contents of Records

change transforms the records passing through in a way similar to the operation of the CHANGE XEDIT subcommand, replacing occurrences of one string with another string. As with XEDIT, the strings may be null. If the first string is null, the second one is inserted; if the second string is null, all occurrences of the first one are deleted.



Type: Filter.

Syntax Description: A keyword is optional. An input range or a list with up to ten ranges in parentheses is optional after the keyword; the default is the complete input record. (This is equivalent to the XEDIT zone setting.) When ranges are specified in parentheses, they must be in ascending order and must not overlap; they refer to positions in the input record. The change specification is mandatory; it specifies two strings. These can be specified between three delimiter characters as in XEDIT or as two separate delimited strings, or as binary/hexadecimal literals. A final number is optional to indicate the maximum number of substitutions in a record. When the first string is not null, the default is to change all occurrences. When the first string is null, the second string is inserted only once; in that case, if the number is specified, it must be 1.

Operation: Within each column range (or the complete record if no range is specified), occurrences of the first string are replaced with the second string; they are deleted if the second string is null. Data between substitutions are copied to the output record unchanged. The substituted string is not scanned for occurrences of the string to be changed.

When the first string is null, the second string is inserted in front of the first range in records that extend to the beginning of that range. If specified, the *numorstar* must be

change

one (1). That is, the string is inserted before the contents of the first column of the first range if the record extends into that range; the string is appended to records that end in the position before the first range; short input records are copied unmodified to the output. Thus, null records are replaced with the string when the first range begins with column 1.

When the keyword `ANYCASE` is specified, the characters in the first string and the input record are compared in upper case to determine whether the specified string is present. When the first string contains one or more upper case characters or contains no letters, the second string is inserted in the output record without change of case; otherwise, an attempt is made to preserve the case of the string being replaced. When the first string contains no upper case letters and begins with one or more (lower case) letters, the following rules determine the case of the replacement string:

- When the first two characters of the replaced string in the input record are both lower case, the replacement string is used without change.
- When the first character of the replaced string in the input record is upper case and the second one is lower case (or not a letter or the string is one character), the first letter of the replacement string is upper cased.
- When the first two characters of the replaced string in the input record are upper case, the complete replacement string is upper cased.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Both changed and unchanged records are written to the primary output stream when no secondary output stream is defined. When the secondary output stream is defined, changed records are written to the primary output stream; unchanged records are written to the secondary output stream.

Record Delay: *change* strictly does not delay the record.

Commit Level: *change* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *change* terminates when it discovers that no output stream is connected.

See Also: *insert*.

Examples: To change all occurrences of “this” to “that”:

```
...|change /this/that/|...
```

To prefix a constant to each record:

```
pipe literal abc|change //msg /|console
►msg abc
►Ready;
```

The first string must be entirely within a column range; adjacent ranges are not merged:

```
pipe literal xabab | literal abab | change /ab/cd/ | console
►cdcd
►xcdcd
►Ready;
pipe literal xabab | literal abab | change (1.2 3-*) /ab/cd/ | console
►cdcd
►xabcd
►Ready;
```


Caseless replacement:

```

pipe literal pipe | change anycase /pipe/line/ | console
▶line
▶Ready;
pipe literal Pipe | change anycase /pipe/line/ | console
▶Line
▶Ready;
pipe literal PiPe | change anycase /pipe/line/ | console
▶Line
▶Ready;
pipe literal PIpe | change anycase /pipe/line/ | console
▶LINE
▶Ready;
pipe literal PiPe | change anycase /Pipe/line/ | console
▶line
▶Ready;

```

One or both strings can be specified as a binary or hexadecimal literal:

```

pipe literal 1234 | change /2/ b11000010 | console
▶1B34
▶Ready;
pipe literal 1234 | change xf2 /second/ | console
▶1second34
▶Ready;
pipe literal 1234 | change xf2 x82 | console
▶1b34
▶Ready;

```

To upper case all occurrences of the string “user”, irrespective of its case (for instance, “User”):

```
... | change anycase /user/USER/ | ...
```

To change a string in the second word of the record:

```

pipe literal abcdefghi ghi | literal hx ghi qh | change w2 /h/*/ | ...
... console
▶hx g*i qh
▶abcdefghi g*i
▶Ready;

```

Notes:

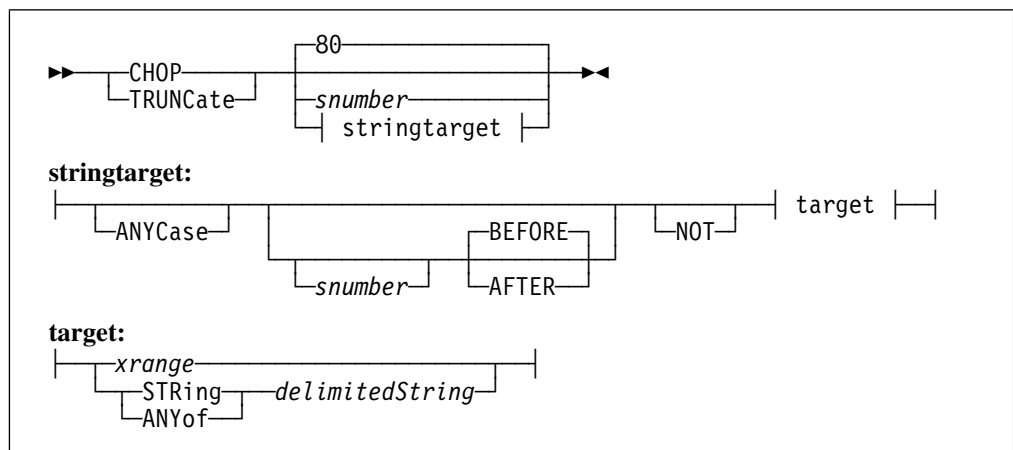
1. XEDIT assumes a null second string when the ending delimiters are omitted; *CMS Pipelines* requires that the change string is specified completely.
2. *change* is similar to the CHANGE XEDIT subcommand, with the extension of multiple ranges (XEDIT supports one range only).
3. *insert* and *spec* can be used to insert a string in all records; *change* with a null first string inserts the second string only in records that contain the column before the first column of the first range.
4. The default is to change all occurrences in the record; the XEDIT default is to change one occurrence only. XEDIT change is case sensitive (even when XEDIT command “case mixed ignore” has been issued); thus XEDIT does not support the function provided by ANYCASE.

chop

5. When the secondary output stream is defined, *change* works as a selection stage, equivalent to *locate* for the first string. In this configuration, *change* discards output records that are written to an unconnected output stream as long as the other output stream is still connected; it terminates prematurely when both output streams are not connected. Unlike a selection stage, *change* writes “unmatched” records to the primary output stream when the secondary output stream is not defined.
6. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
7. Only one *inputRange* can be specified. When specifying a list of ranges in parentheses, the ranges must be old-fashioned column ranges, even when there is only one range in the parentheses.
8. *change* cannot be cajoled into appending a string at the end of the record in general. Use *insert* instead.

chop—Truncate the Record

chop truncates records after a specified column, or at a specified character or string.



Type: Filter.

Syntax Description: With no arguments, input records are truncated after column 80. With a single operand that is zero or positive, *chop* truncates the record after this column; the result of truncating after column zero is a null record. With a single negative operand, *chop* truncates the record after the column that is the sum of the record length and the negative operand; the result of truncating before column zero is a null record (that is, the record is shorter than the negative of the operand).

Use a hex range or a delimited string to truncate the record depending on its contents. A hex range matches any character within the range. The keyword `STRING` followed by a delimited string matches the string. The keyword `ANYOF` followed by a delimited string matches any one character in the string. (The keyword is optional before a one character string, because the effect is the same in either case.) The default is to truncate the record before the first character matching the target.

ANYCASE	Ignore case. Conceptually, all processing is done in upper case.
NOT	Truncate the record relative to a character or string that does not match the specified target.

The truncation column may be specified with an offset relative to the beginning or end of the matching string or character. The offset may be negative.

Operation: A chop position is established in the input record. Data, if any, before the chop position are written to the primary output stream; the remainder of the input record is written to the secondary output stream (if it is defined and connected).

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Output is written to the primary output stream and the secondary output stream.

Record Delay: *chop* strictly does not delay the record.

Commit Level: *chop* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *chop* terminates when it discovers that no output stream is connected.

See Also: *join*, *spill*, *split*, and *strip*.

Examples: To truncate the record before the first blank:

```
! pipe literal sample line | chop blank | console
! ▶sample
! ▶Ready;
```

```
! There is no MODULO option on chop to truncate the records to the largest multiple of a
! specified modulo, but vchar can do that. For example to truncate to a multiple of 4 bytes
! (32 bits):
```

```
! pipe xrange 0 9 | vchar 32 32 | console
! ▶01234567
! ▶Ready;
```

```
! The record of 10 bytes is truncated to 8; the largest multiple of 4 that is not larger than 10.
```

Notes:

1. “chop not z | locate 1” discards all records without leading z(s) because *chop* truncates before the first character that is not a z. This yields a null record, which is discarded by *locate*.
2. Records are not padded.
3. The minimum abbreviation of ANYCASE is four characters because ANYOF takes precedence (ANYOF can be abbreviated to three characters).
4. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
5. -0 is not a signed number according the *CMS Pipelines* scanning rules.
6. The ambiguity of a single digit as the only operand is resolved as a signed number rather than matching a range of a single character.

cipher

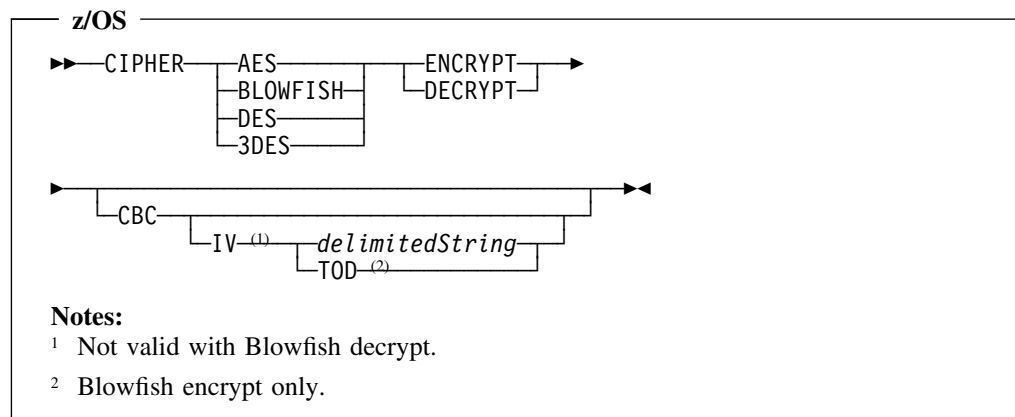
cipher—Encrypt and Decrypt Using a Block Cipher

cipher encrypts or decrypts according to the following algorithms:

- Advanced Encryption Standard (AES).
- Blowfish, Bruce Schneier’s algorithm.
- Data Encryption Standard (DES).
- Triple DES (3DES).

Electronic Code Book (ECB) and Cipher Block Chaining (CBC) modes are supported.

The key is supplied as an input record.



Type: Filter.

Syntax Description: The first word specifies the algorithm to be used. The second word specifies the “direction”.

AES	Use the Advanced Encryption Standard algorithm.
BLOWFISH	Use the Blowfish algorithm.
DES	Use the Data Encryption Standard algorithm when an 8 byte key is supplied; use triple DES when a 16 or 24 byte key is supplied.
3DES	Use the triple Data Encryption Standard algorithm. Two or three 8 byte keys (16 or 24 bytes) are used. The keys must not all be equal.
CBC	Select cipher block chaining mode. The default is electronic code book mode.
IV	Specify the initialisation vector for cipher block chaining mode. The contents of the delimited string are repeated as necessary to fill a block. The keyword TOD (Blowfish encrypt only) selects the sixty-four bit contents of the time-of-day clock. No initialisation vector is specified with Blowfish decrypt, as it is received as the first cipher block.

Operation:

Figure 388. Summary of Architectures.

	AES	Blow	DES	3DES
Uses hardware instructions	Yes	N/A	Yes	Yes
Fall back to software when hardware instruction is not available	No	Yes	Yes	Yes
Block size in bytes	16	8	8	8
Key length in bytes	16, 24 or 32	Any	8, 16, or 24	16 or 24

For Blowfish, the initial vector is encrypted and written to the output; the decrypting stage decrypts this initial record and writes the initialisation vector in plain text to the output. This first record would normally be discarded when decrypting.

Key Handling: When no secondary streams are defined, the first record on the primary input stream is used as the key; there is then no provision for dynamic key change.

When the secondary input stream is defined, the first record is read unconditionally from that stream and used as the initial key. The pipeline will stall if a record is presented on the primary input stream instead.

The key is changed dynamically when a record that is not null is read from the secondary input.

Input Record Format: Input records must have a length that is a multiple of the block size.

Streams Used: Secondary streams may be defined. Records are written to the primary output stream; no other output stream may be connected. Null input records are discarded.

Record Delay: *cipher* does not delay the record.

Commit Level: *cipher* starts on commit level -2. It verifies that the primary output stream is the only connected output stream and then commits to level 0.

Premature Termination: *cipher* terminates when it discovers that its primary output stream is not connected.

Converse Operation: *cipher*.

See Also: *digest*.

Examples: A famous test case:

ckddeblock

```
/* Cipher sample from FIPS 140-2 */
Signal on novalue
Address COMMAND
'PIPE (end \ name CPHRSAMP)',
  '\literal Now is the time for all',/* Trailing blank */
  '|xlate e2a',
  '|strliteral x0123456789abcdef', /* Key */
  '|cipher des encrypt',
  '|deblock 16',
  '|spec 1.4 c2x 1 5.4 c2x nw 9.4 c2x nw 13.4 c2x nw',
  '|cons'
Exit RC

cphrsamp
▶3FA40E8A 984D4815 6A271787 AB8883F9
▶893D51EC 4B563B53
▶Ready;
```

Notes:

1. “Hardware instructions” should be taken to mean “Message-security Assist” and “Message-security Assist Extension 1” and “Message-security Assist Extension 2” facilities. *cipher* specifically does not support Cryptographic coprocessors (“Integrated cryptographic facility”).
2. The CP assist feature must be installed and enabled for the underlying instructions to be available in a virtual machine.
3. The user should ensure that there is never a record present at the same time on both input streams, as this would lead to an indeterminate time of key change, which, in general, would make the enciphered text indecipherable.
4. When DES is specified and a 16-byte key is used, the first eight bytes should be different from the last eight bytes (if not, we have single DES); however, the triple DES standard specifies to use such a key to interoperate with single DES; thus, this is not enforced. The performance increase by downgrading to single DES is not exploited.
5. When BLOWFISH is specified, the initial S and P boxes cannot be changed from the default; see <http://www.schneier.com/code/constants.txt>
6. Use *pad* MODULO to pad records out to full cipher blocks.

ckddeblock—Deblock Track Data Record

ckddeblock splits a track data record into its three parts:

- Count area (CCHHRKDD).
- Key area. A null record is written when the key length (K) is zero in the count area.
- Data area. A null record is written when the data length (DD) is zero in the count area.

▶▶—CKDDEBLOCK—◀◀

Type: Arcane filter.

Record Delay: *ckddeblock* does not delay the record.

Premature Termination: *ckddebloc* terminates when it discovers that its output stream is not connected.

Notes:

1. The converse operation is *join 2*.
2. For ECKD devices with more than 65519 cylinders, Extended Address Volumes format specifies how a 28-bit cylinder number and 4-bit track number are encoded in the 32-bit word referred to as CCHH.

cms—Issue CMS Commands, Write Response to Pipeline

cms issues CMS commands with full command resolution, and captures the command response, which is then written to the output of the stage rather than being displayed on the terminal.



Type: Host command interface.

Syntax Description: A string is optional. No argument is allowed when the secondary output stream is defined.

Operation: The argument string (if present) and input lines are issued to CMS through the CMS subcommand environment, as REXX does for the Address CMS instruction.

The response from the CMS commands is not written to the terminal. The response from each command is buffered until the command ends and is then written to the primary output stream. *cms* does not intercept CP-generated terminal output.

Each invocation of *cms* maintains a private CMSTYPE flag; this flag is initially set as it is by SET CMSTYPE RT. If a command that is issued through a particular invocation of *cms* issues SET CMSTYPE HT, subsequent command response lines that apply to the stage are discarded until a SET CMSTYPE RT command is issued while the stage is running. The HT/RT setting is preserved between commands.

When the secondary output stream is defined, the return code is written to this stream after each command has been issued and the response has been written to the primary output stream.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded. The response of the command is written to the primary output stream. The return code of each command is written to the secondary output stream when connected.

Record Delay: *cms* writes all output for an input record before consuming the input record. When the secondary output stream is defined, the record containing the return code is written to the secondary output stream with no delay.

Commit Level: *cms* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

collate

Premature Termination: When the secondary output stream is not defined and *cms* receives a negative return code on a command, it terminates. The corresponding input record is not consumed. When the secondary output stream is defined, *cms* terminates as soon as it discovers that this stream is not connected. If this is discovered while a record is being written, the corresponding input record is not consumed.

See Also: *aggrc*, *command*, *cp*, *starmsg*, *subcom*, and *tso*.

Examples: To discard the service level information in the CMS version message:

```
pipe cms query cmslevel | chop , | console
►CMS Level 28
►Ready;
```

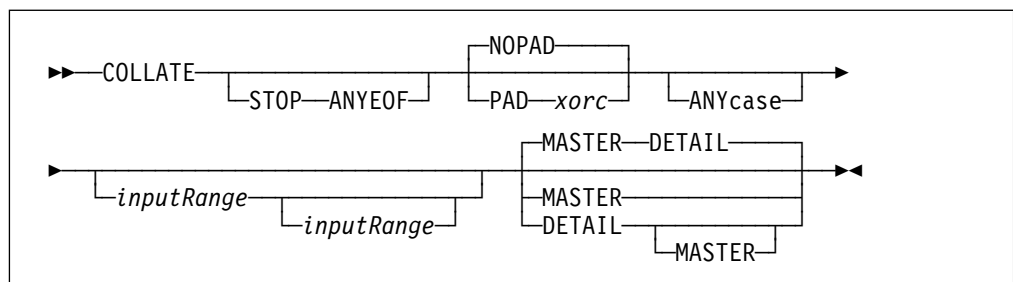
Notes:

1. Use *subcom* CMS to issue CMS commands without intercepting line mode output to the terminal.
2. *cms* is not recommended to invoke applications that run in full screen mode, for instance, XEDIT, because line mode console output is intercepted. Any line mode output during the session (for instance, REXX error messages) is delayed until the application completes.
3. Do not issue the immediate commands HT and RT while a *cms* stage is dispatched; this action cannot be distinguished from the SET CMSTYPE command.
4. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: When a secondary output stream is not defined and a negative return code is received on a command, the return code from *cms* is that negative return code. When a secondary output stream is not defined and the return code is zero or positive, all input records have been processed; the return code is the maximum of the return codes received. When the secondary output stream is defined, the return code is zero unless an error is detected by *cms*.

collate—Collate Streams

collate compares two input streams containing master records and detail records. Depending on the contents of a key field in the records, input records are passed to one of three output streams (if connected) or discarded.



Type: Sorter.

Syntax Description: The keyword NOPAD specifies that key fields that are partially present must have the same length to be considered equal; this is the default. The keyword PAD specifies a pad character that is used to extend the shorter of two key fields.

The keyword ANYCASE specifies that case is to be ignored when comparing fields; the default is to respect case.

Two input ranges are optional; the default is the complete record. The first input range defines the key on the primary input stream; the second input range defines the key on the secondary input stream. When both ranges are specified, any WORDSEPARATOR or FIELDSEPARATOR specified for the first input range applies to the second input range as well, unless specified again. A single input range applies to both input streams.

Two keywords are optional to define the sequence of records on the primary output stream.

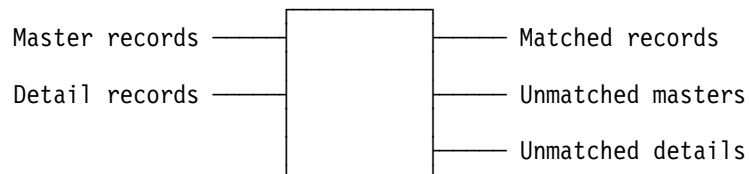
Operation: The master file is read from the primary input stream; detail records are read from the secondary input stream. Both files should be ordered ascending by their keys. The master file should only have one record for each key.

Three output streams are written (if connected):

- 0 Master and detail records when both are present for a key. The order of records is defined by the keywords. The default is to write the master record followed by all detail records referring to the particular key. Specify one or two keywords to select which type of record to write and the order to write them in. Records are discarded if you specify only one keyword.
- 1 Master records for which there is no corresponding detail record.
- 2 Detail records for which there is no corresponding master record.

Streams Used: Two or three streams may be defined. If it is defined, the tertiary input stream must not be connected. Records are read from the primary input stream and secondary input stream. Records are written to all defined output streams.

Unless STOP ANYEOF is specified, the primary input stream is shorted to the secondary output stream when the secondary input stream reaches end-of-file and the secondary input stream is shorted to the tertiary output stream when the primary input stream reaches end-of-file. When STOP ANYEOF is specified *collate* terminates as soon as it senses end-of-file on either input stream. The other stream is left unconsumed.



Record Delay: *collate* strictly does not delay the record.

Commit Level: *collate* starts on commit level -2. It verifies that the tertiary input stream is not connected and then commits to 0.

Premature Termination: *collate* terminates when it discovers that no output stream is connected.

See Also: *lookup*, *merge*, and *sort*.

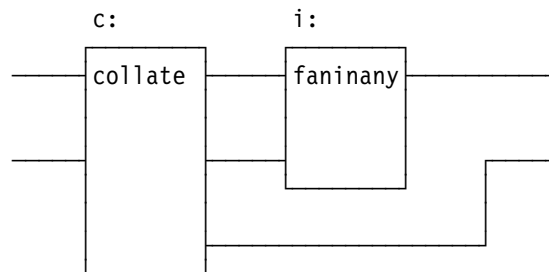
collate

Examples: Assuming that the file STOP WORDS contains an ordered list of words to suppress, this subroutine pipeline suppresses such words from the caller's input stream:

```
/* Do stop words */
'callpipe (end ?) :* | split | sort unique | c: collate',
                        '?< stop words | c:          | *:'
```

To include master records that have no corresponding detail records in the primary output stream:

```
/* Allow unreferenced master records */
'callpipe (end ?)',
 '*input.0: | c: collate' arg(1) ' | i: faninany | *.output.0:',
 '?*.input.1: | c:                  | i:',
 '?          c:                  | *.output.1:'
Exit RC
```



To prefix a field from the corresponding master record to each detail record and discard unmatched master records. Unmatched detail records are written to the secondary output stream:

```
/* Collate master in front of detail */
'callpipe (end ? name COLLATE)',
 '*input.0:', /* Master file */ /*
 '|o:fanout', /* Make copy */ /*
 '|spec 20.10 1', /* Select field */ /*
 '|j:juxtapose', /* Store for later */ /*
 '*output.0:', /* Write details with prefix */ /*
 '?o:', /* Master records */ /*
 '|c: collate 1.10 detail', /* Collate with details */ /*
 '|j:', /* Go prefix field */ /*
 '?*.input.1:', /* Detail records */ /*
 '|c:', /* Collate them */ /*
 '?c:', /* Unmatched details here */ /*
 '*output.1:' /* Pass them on */ /*
```

To fill out holes in a sequence:

```

.      /* Fill out holes in a sequence                                */
.      Signal on novalue
.      Address COMMAND
.      'PIPE (end ? name FILLOUT)',
.          '?literal 1 3 5',          /* Some test data          */
.          '|split',                 /* Make three records     */
.          '|pad left 10',           /* Shift right            */
.          '|c: collate stop anyeof 1.10 master',
.          '|i: faninany',          /* Add fillers            */
.          '|cons',                 /* Show them              */
.          '?literal',              /* One record             */
.          '|dup *',                /* Infinitely many        */
.          '|spec number 1',        /* Generate all members of sequence */
.          '|c:',                  /* Pass to collate        */
.          '|?c:',                 /* Unmatched details     */
.          '|insert / Missing./ after', /* Mark them              */
.          '|i:'                   /* Add to main file       */
.      Exit RC
.
.      fillout
.      ▶          1
.      ▶          2 Missing.
.      ▶          3
.      ▶          4 Missing.
.      ▶          5
.      ▶R;

```

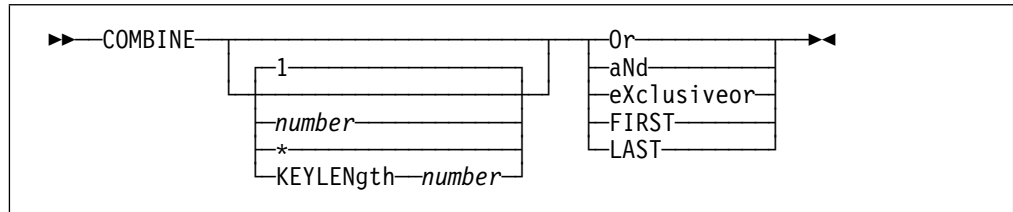
Notes:

1. Unless ANYCASE is specified, key fields are compared as character data using the IBM System/360 collating sequence.
2. Use *spec* (or a REXX program) for example to put a sort key in front of the record if you wish, for instance, to use a numeric field that is not aligned to the right within a column range. Such a temporary sort key can be removed with *substr* for example after the records are written by *collate*.
3. Use *xlate* to change the collating sequence of the file.
4. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
5. *collate* supports only one key field (unlike *sort*). Use *spec* to gather several key fields into one. When this temporary key is placed in front of the original record, it can be removed from the output of *collate* easily with *substr* even when the input records vary in length or fields get added at some later time.

combine—Combine Data from a Run of Records

combine combines the contents of several input records into one output record. The contents of an output column are a function of the contents of that particular column in a range of input records. The function can be inclusive OR, AND, exclusive OR; or it can select the first or last record that contains data for each column. Combining two records with exclusive OR generates a map of characters that are identical in the two records.

When only one input stream is defined, *combine* combines runs of records on this stream. When two input streams are defined, *combine* combines pairs of records, one from each stream.



Type: Filter.

Syntax Description: When only one input stream is defined, a number is optional; it specifies how many records to combine with the first one in a range. (That is, **one less** than the number of records in a range.) The default is to combine two input records (as if the number 1 were specified). Specify an asterisk to combine all input records into one output record.

KEYLENGTH specifies that runs of records that contain the same key in the first *n* columns are combined. (*n* is the number specified after the keyword.)

When two input streams are defined, a number or KEYLENGTH is rejected. The keyword STOP specifies when *combine* should terminate. ALLEOF, the default, specifies that *combine* should continue as long as at least one input stream is connected. ANYEOF specifies that *combine* should stop as soon as it determines that an input stream is no longer connected. A number specifies the number of unconnected streams that will cause *combine* to terminate. The number 1 is equivalent to ANYEOF.

A keyword specifies the operation to perform when combining records:

- O Bitwise inclusive OR. A bit in the output record is 1 if any record in the range has that particular bit on; it is 0 when all input records in the range have that particular bit off.
- N Bitwise AND. A bit in the output record is 0 if any record in the range has that particular bit off; it is 1 when all input records in the range have that particular bit on.
- X Bitwise exclusive OR. A bit in the output record is 0 if an even number of records in the range have that particular bit on (or the bit is not on in any input record in the range); the bit is 1 when an odd number of input records in the range have that particular bit on.
- FIRST The contents of an output column are taken from the first record in the range that contains the particular column.
- LAST The contents of an output column are taken from the last record in the range that contains the particular column.

Operation: *combine* supports a single input stream and two input streams.

When a single input stream is defined, runs of records are combined. When two input streams are defined, a record from the primary input stream is combined with a record from the secondary input stream.

Records are combined as follows: A buffer to build the output record is made empty. Each input record has two (possibly null) parts, the part that corresponds to buffer positions that have been filled by previous records in the range, and the part by which the record exceeds the contents of the buffer so far. The latter part is appended to the contents of the buffer; the first part, if any, is processed according to the particular function

requested. The contents of the buffer are written to the output when all records in the range have been processed or when end-of-file is met.

When KEYLENGTH is specified, the key is left unchanged; only positions beyond the key are combined. When there is only one record with a particular key, it is copied unchanged to the output.

Record Delay: When *combine* combines records from two input streams, it strictly does not delay the record.

When it combines records from the primary input stream, *combine* writes the output record before it consumes the last record in a run of combined records. Thus, the last record of a run is not delayed; records before the last one are discarded.

Premature Termination: *combine* terminates when it discovers that its primary output stream is not connected.

Examples: Given two variables, before and after, determine how many columns contain the same character in the two records (assuming they do not contain X'00'):

```
/* Now see how many */
'callpipe (name COMBINE)',
  | var before',
  | append var after',
  | combine x',
  | xlate *- * 01-ff 1 00 0',          /* byte-map of inequality */
  | deblock 1',
  | sort count',
  | ...
```

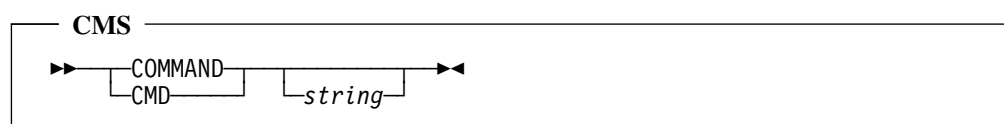
Assuming that the two input records are the same length, each byte of the output record from *combine* contains the exclusive OR of the bytes in the corresponding positions of the input records. A byte that contains X'00' indicates that the two input bytes were equal; a nonzero value shows the bit differences. The *xlate* stage maps a byte of zeros to the character "0" and the 255 other possible values to the character "1". Thus, at the output from *xlate*, each position of the record contains a zero if the input records were equal and a one if they were not. *deblock* writes a record for each character in its input record and *sort COUNT* computes the distribution.

Notes:

1. The option O can be written in full: OR. AND is a synonym for N. EXCLUSIVEOR is a synonym for X; it can be abbreviated down to three characters.
2. Input records of different lengths are not padded; rather, the last part of the longer record is copied to the output record without modification.

command—Issue CMS Commands, Write Response to Pipeline

command issues CMS commands that can be resolved to modules or CMS nucleus routines and captures the command response, which is then written to the output of the stage rather than being displayed on the terminal.



command

Type: Host command interface.

Syntax Description: A string is optional. No argument is allowed when the secondary output stream is defined.

Operation: The argument string (if present) and input lines are issued as CMS commands using program call with an extended parameter list, as REXX does for the address command instruction.

The command is passed to CMS using CMSCALL with a call flag byte of 1 indicating that an extended parameter list is present (but not command call). The argument string and input lines should be in upper case unless you wish to manipulate objects with mixed case names.

The response from the CMS commands is not written to the terminal. The response from each command is buffered until the command ends and is then written to the primary output stream. *command* does not intercept CP-generated terminal output.

Each invocation of *command* maintains a private CMSTYPE flag; this flag is initially set as it is by SET CMSTYPE RT. If a command that is issued through a particular invocation of *command* issues SET CMSTYPE HT, subsequent command response lines that apply to the stage are discarded until a SET CMSTYPE RT command is issued while the stage is running. The HT/RT setting is preserved between commands.

When the secondary output stream is defined, the return code is written to this stream after each command has been issued and the response has been written to the primary output stream.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded. The response of the command is written to the primary output stream. The return code of each command is written to the secondary output stream when connected.

Record Delay: *command* writes all output for an input record before consuming the input record. When the secondary output stream is defined, the record containing the return code is written to the secondary output stream with no delay.

Commit Level: *command* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: When the secondary output stream is not defined and *command* receives a negative return code on a command, it terminates. The corresponding input record is not consumed. When the secondary output stream is defined, *command* terminates as soon as it discovers that this stream is not connected. If this is discovered while a record is being written, the corresponding input record is not consumed.

See Also: *aggrc*, *cms*, *cp*, *starmsg*, *subcom*, and *tso*.

Examples: To discard the service level information in the CMS version message:

```
/* Show CMS version without service level */  
pipe command QUERY CMSLEVEL | chop , | console  
►CMS Level 28  
►Ready;
```

command is useful to manipulate file objects with names in mixed case. To erase the file “mIxEd CaSe”:

```
pipe command ERASE mIxEd CaSe A | console
```

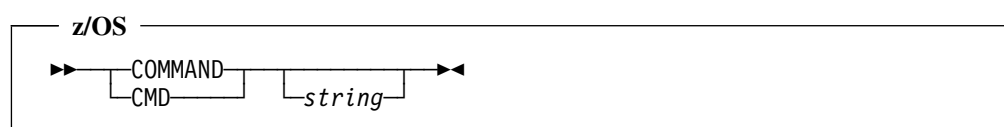
Notes:

1. Use *subcom* CMS to issue CMS commands without intercepting line mode output to the terminal. Use *cms* to issue CMS commands with full command resolution.
2. *command* is not recommended to invoke applications that run in full screen mode, for instance, XEDIT, because line mode console output is intercepted. Any line mode output during the session (for instance, REXX error messages) is delayed until the application completes.
3. Do not issue the immediate commands HT and RT while a *command* stage is dispatched; this action cannot be distinguished from the SET CMSTYPE command.
4. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: When a secondary output stream is not defined and a negative return code is received on a command, the return code from *command* is that negative return code. When a secondary output stream is not defined and the return code is zero or positive, all input records have been processed; the return code is the maximum of the return codes received. When the secondary output stream is defined, the return code is zero unless an error is detected by *command*.

command—Issue TSO Commands

command issues TSO commands. The command response is written to the terminal by TSO.



Type: Host command interface.

Syntax Description: A string is optional.

Operation: The argument string (if present) and input lines are passed to the TSO service routine to be issued as commands. A return code from this service routine indicating that the command does not exist is recoded as return code -3 from the command; other errors from the service routine cause processing to terminate with an error message.

When the secondary output stream is defined, the return code is written to this stream after each command has been issued.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded. *command* does not write to the primary output stream. If the secondary output stream is defined, the return code is written to it.

configure

Record Delay: When the secondary output stream is defined, the record containing the return code is written to the secondary output stream with no delay.

Commit Level: *command* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: When the secondary output stream is not defined and *command* receives a negative return code on a command, it terminates. The corresponding input record is not consumed. When the secondary output stream is defined, *command* terminates as soon as it discovers that this stream is not connected. If this is discovered while a record is being written, the corresponding input record is not consumed.

See Also: *subcom* and *tso*.

Examples: To issue a command from a REXX filter (which is not merged with the TSO environment and therefore has no ability to Address TSO):

```
/* Now do the command */  
'callpipe var command | command'
```

Notes:

1. Use *tso* to issue TSO commands and write the response to the pipeline for further processing.
2. *command* issues GCS commands on GCS.
3. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: When a secondary output stream is not defined and a negative return code is received on a command, the return code from *command* is that negative return code. When a secondary output stream is not defined and the return code is zero or positive, all input records have been processed; the return code is the maximum of the return codes received. When the secondary output stream is defined, the return code is zero unless an error is detected by *command*.

configure—Set and Query CMS Pipelines Configuration Variables

configure accesses and sets CMS Pipelines configuration variables.

When *configure* is first in a pipeline, it writes to the output stream the value of all configuration variables that have been set.

When *configure* is not first in a pipeline, its input records contain the name and optionally the new value for configuration variables. *configure* updates the variable (if a second word is specified) and then writes the value of the specified variable to the output stream.

►►—CONFIGURE—◄◄

Type: Service program.

Input Record Format: Each record may contain one or two words. The first word is the name of the configuration variable. Case is ignored in variable names.

If the second word is present, it contains a new value for the specified configuration variable. Case is ignored in keywords; values are made upper case.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *configure* does not delay the record.

Examples:

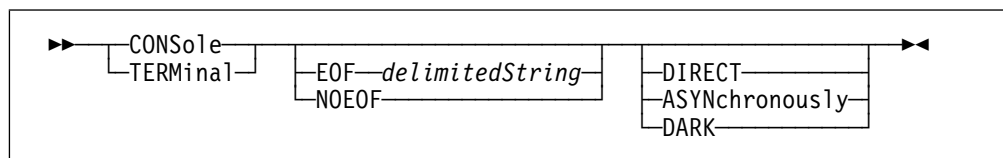
```
pipe literal style | configure | console
▶STYLE DMS
▶Ready;
```

Notes:

1. Refer to Chapter 28, “Configuring *CMS Pipelines*” on page 867 for a list of configuration variables.

console—Read or Write the Terminal in Line Mode

When *console* is first in a pipeline it reads lines from the terminal and writes them into the pipeline. When *console* is not first in a pipeline it copies lines from the pipeline to the terminal.



Type: Device driver.

Syntax Description: No argument is allowed when *console* is not first in a pipeline. Keywords are optional when *console* is first in a pipeline.

EOF specifies a delimited string; end-of-file is signalled when this string is entered (with leading or trailing blanks, or both). NOEOF specifies that input data are not inspected for an end-of-file indication; *console* stops only when it finds that its output stream is not connected. The null string signals end-of-file if neither of these keywords is specified.

A second type of keyword is supported only on CMS. It specifies the interface to be used when reading from the terminal. If the keyword is omitted, a normal CMS terminal read is performed; the program stack and the console queue are emptied before CMS reads from the terminal, at which time a VM READ is put up on the virtual machine console.

When one of the second type of keywords is specified, *console* performs a *direct read*; that is, *console* reads directly from the terminal. The program stack and the console queue are bypassed.

console

DIRECT	Standard direct reads are performed as described above. A VM READ is issued immediately.
ASYNCHRON	Direct reads are issued in response to attention interrupts. This means that the console is not locked in a VM READ while waiting for user input.
DARK	“Invisible” direct reads are issued. A VM READ is issued immediately. The input is not echoed to the upper part of the terminal.

Operation: When *console* is first in a pipeline, lines are read from the terminal until a line is read that is equal to the delimited string specified with EOF (by default the null string).

When *console* is not first in a pipeline, lines from the primary input stream are written to the terminal of the virtual machine, and copied to the primary output stream, if connected.

When *console* is in a pipeline set that has been issued under control of *runpipe* EVENTS, *console* signals console read or write events, as appropriate, instead of accessing host interfaces.

Streams Used: When *console* is first in a pipeline (it is reading from the terminal), records are written to the primary output stream. When *console* is not first in a pipeline (it is writing to the terminal), records are read from the primary input stream and copied to the primary output stream, if connected.

Record Delay: *console* strictly does not delay the record.

Premature Termination: When it is first in a pipeline, *console* terminates when it discovers that its output stream is not connected.

Examples: To read lines directly from the terminal into the stack without going into a loop:

```
pipe console direct | stack
```

To read lines until a line consisting of just two blanks is entered:

```
pipe console eof x4040 | ...
```

Notes:

- !
 - !
 - !
 - !
 - !
1. Only one *console* stage should be used to read from the console at any time as it is unspecified which of the active *console* stages would read the record. This is not enforced by *CMS Pipelines*.
 2. *console* ASYNCHRONOUSLY should be used with caution; it is not possible to enter CMS immediate commands while it is waiting for terminal input.
 3. One *console* ASYNCHRONOUSLY stage can read from the terminal at a time; a subsequent one stacks the current reading stage, which resumes control of the terminal when the new stage terminates.
 4. On z/OS, GETLINE and PUTLINE macros are used to read and write the TSO terminal. TGET and TPUT are used when the pipeline is started with the CALL command or referenced with PGM= in an EXEC job control statement and a TSO environment is active. Write to programmer is used as a last resort.

5. Lines written to the terminal are truncated to fit the particular interface on z/OS. No truncation is required for CMS. (But CP may truncate the lines written to the console SPOOL.)
6. Use the CMS command PIPMOD STOP or send a record into the *pipestop* stage to terminate *console* ASYNCHRONOUSLY while it is waiting for an attention interrupt. Note that you cannot enter immediate commands from the terminal while *console* ASYNCHRONOUSLY is running; the command must be generated in the pipeline.
7. On input, CMS has performed the SET INPUT translation on the lines typed at the console before *console* reads them and writes them to the pipeline. Likewise, CMS does SET OUTPUT translation after *console* writes the line to the terminal.
8. *terminal* is a synonym for *console*. INVISIBLE is a synonym for DARK.
9. On CMS input is truncated after 1024 characters. Note that the stack still truncates at 255. Thus, longer lines must be typed at the terminal (and the terminal must have a suitably long input area).

copy—Copy Records, Allowing for a One Record Delay

copy passes the input to the output in a way that can delay by one record. It may be useful to avoid a stall in a pipeline network where a delay of one record is sufficient to prevent the stall.

►►—COPY—◄◄

Type: Arcane filter.

Operation: Each input record is read into a buffer. The input record is consumed before the contents of the buffer are written to the output.

Record Delay: *copy* has the potential to delay one record.

Premature Termination: *copy* terminates when it discovers that its output stream is not connected.

See Also: *elastic*.

Examples: To hold the output on the primary output stream from *chop* while the balance of the input record is being written to the secondary output stream, and the two pieces are to be reunited in a later stage:

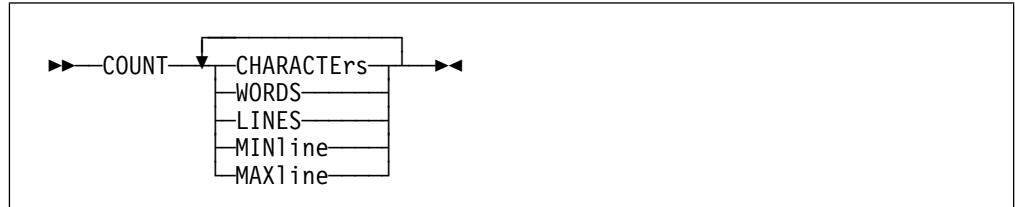
```
/* Uppercase label: */
'PIPE (end ? name COPY)',
    ... ,
    '|c: chop before blank',
    '|  xlate upper',
    '|  copy',
    '|s: spec 1-* 1 select 1 1-* next',
    ... ,
    '?c:',
    '|s:'
```

Records that have a leading blank (that is, assembler statements that have no label field) are passed in their entirety to the secondary output stream after a null record has been written to the primary output stream.

count

count—Count Lines, Blank-delimited Words, and Bytes

count counts the number of input lines, words, characters, or any combination thereof. It can also report the length of the shortest or longest record, or both. It writes a line with the specified counts at end-of-file.



Type: Filter.

Syntax Description: Specify what to count; you can specify up to five keywords in any order.

Operation: Input records are read and counters are updated; the counters are 56 bits ignoring overflow. Bytes and lines are counted without reference to the contents of the input record; *count* WORDS references the storage area holding the input record.

When there are no input records, the shortest record is reported as infinity (2G-1, because this is the longest possible record); the longest record is reported as null.

Output Record Format: A record is built with the result when the primary input stream reaches end-of-file. Irrespective of the order of the options, this record has a number for each specified counting option in the order characters, words, lines, minimum record length, and maximum record length. There is one blank between numbers.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. When the secondary output stream is not defined, input records are discarded; the record containing the counts is written to the primary output stream. When the secondary output stream is defined, the input records are copied to the primary output stream; the primary output stream is severed at end-of-file on the primary input stream; and the record containing the counts is then written to the secondary output stream.

Record Delay: When the secondary output stream is defined, *count* does not delay the records that are copied to the primary output stream.

Premature Termination: If the secondary output stream is defined, *count* terminates when the primary output stream becomes not connected; the counts are written to the secondary output stream; the record that receives end-of-file on the primary output stream is not included in the counts. Thus, the counts on the secondary output stream reflect the amount of data consumed by the stage connected to the primary output stream.

Examples: To count the number of blank-delimited words in the profile:

```
pipe < profile exec | count words | console
▶164
▶Ready;
```

To count both the number of words and the number of unique words:

```

!          /* Special counting */
!          'PIPE (end ?)',
!              '? < profile exec',
!              '| c: count words',
!              '| split',
!              '| sort unique',
!              '| count lines',
!              '| spec 1-* 1 , unique words., next',
!              '| console',
!              '? c:',
!              '| spec 1-* 1 , words total., next',
!              '| console'
!

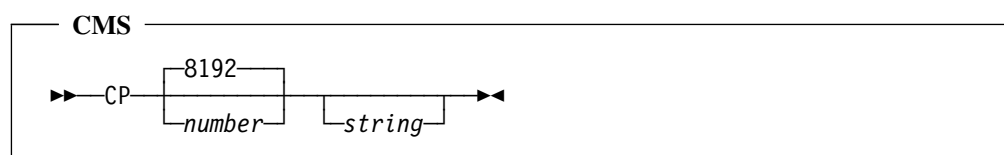
```

Notes:

1. RECORDS is a synonym for LINES. CHARS and BYTES are synonyms for CHARACTERS.

cp—Issue CP Commands, Write Response to Pipeline

cp issues CP commands and captures the command response, which is then written to the output of the stage rather than being displayed on the terminal.



Type: Host command interface.

Syntax Description: A number and a string are optional. If the first word of the argument string is a number, it specifies the required size of the response buffer; the default is 8192. Only a number is allowed when the secondary output stream is defined.

Operation: The argument string (after the number, if present) and input lines are issued as CP commands through the extended diagnose 8 interface. *cp* terminates with an error message if a command is longer than the 240 bytes supported by CP.

The first blank-delimited word of each command is inspected. If it is different from its upper case translation, the **entire** command is translated to upper case before it is issued to CP.

The response from CP is transformed into lines that are written to the primary output stream; lines with length zero are discarded.

When the length of the response buffer is not specified and the command issued is QUERY (or an abbreviation thereof, down to one character), the buffer is extended dynamically to accept the complete command response. In other cases, there is no indication of an error when CP truncates the response because the buffer is too small, unless there is a secondary output stream defined.

When the secondary output stream is defined, the return code is written to this stream after each command has been issued and the response has been written to the primary output stream. The return code has a leading plus sign if the command response was truncated.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded. Secondary streams may be defined.

Record Delay: *cp* writes all output for an input record before consuming the input record. When the secondary output stream is defined, the record containing the return code is written before the corresponding input record is consumed.

Commit Level: *cp* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: When a secondary output stream is not defined, *cp* terminates as soon as it receives return code 1 (unknown command) from CP. The corresponding input record is not consumed. When the secondary output stream is defined, *cp* terminates as soon as it discovers that this stream is not connected. If this is discovered while a record is being written, the corresponding input record is not consumed.

See Also: *aggrc*, *cms*, *command*, *starmsg*, and *subcom*.

Examples: To process all reader files in a query:

```
pipe cp query reader * all | ...
```

cp allocates a sufficiently large buffer to accommodate whatever reply you will receive to the query.

To transfer all reader files to another user, making certain that all 9999 possible SPOOL files can be transferred and the full response still be captured:

```
pipe cp 999900 transfer rdr all to someuser | ...
```

Notes:

1. Write the CP command verb in upper case to avoid translation of the arguments to upper case when using *cp* to issue commands with mixed case arguments. For example:

```
pipe cp MSG OSCAR Hi, there...
```

2. Specify a buffer size operand to issue CP QUERY commands that have side effects (if any exist). This ensures that the command is issued only once.

3. Be careful when using *cp* to store the value of a CP variable in a REXX variable, as in the REXX statement below.

```
'PIPE cp query variable runmode | var runmode'
```

When the CP variable is not set, CP will output an empty line that is discarded by *cp*. When *var* does not get an input record, the specified REXX variable is dropped. Reference to that REXX variable will raise a novalue condition, when enabled.

To ensure the REXX variable is set by the command, consider providing a default value, for example with a *strliteral* stage with IFEMPTY. In the following pipeline the REXX variable runmode will be set to "NORMAL" when *cp* does not produce an output record.

```
'PIPE cp q var runmode | strliteral ifempty /NORMAL/ | var runmode'
```

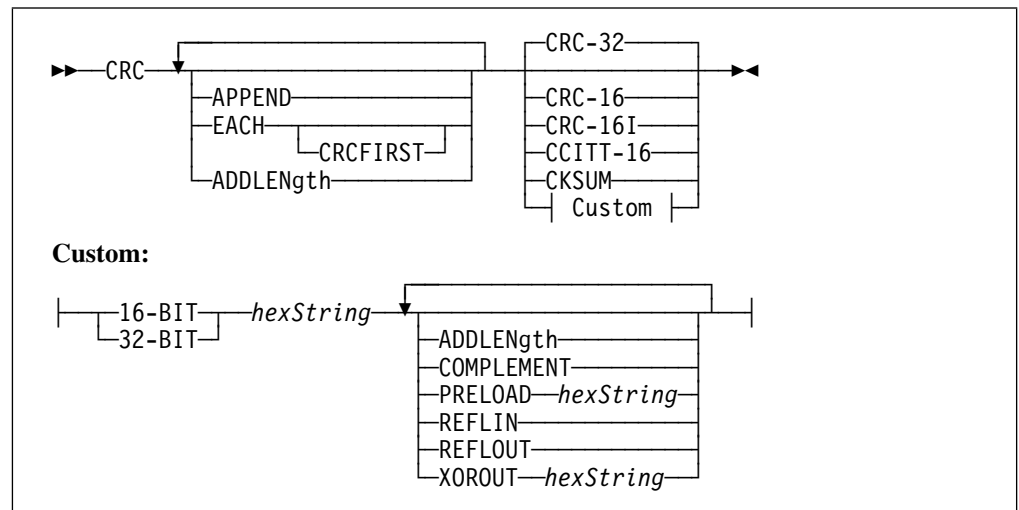
4. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the

comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: When no secondary output stream is defined and *cp* terminates because CP does not recognise a command, the return code is 1 (irrespective of return codes from other commands). When no secondary output stream is defined and the return code is not 1, all input records have been processed; the return code is the maximum of the return codes received from CP. When the secondary output stream is defined, the return code is zero unless *cp* detects an error.

: *crc*—Compute Cyclic Redundancy Code

: *crc* computes the Cyclic Redundancy Code (CRC) of a message.



Type: Filter.

Syntax Description: The operands to *crc* are of two types, general flags, and operands to specify the algorithm.

The general flags are:

- APPEND Pass the input to the primary output. Write the CRC as a separate final record. Secondary streams are not allowed.
- EACH Write the CRC for each record and reset to initial conditions.
- CRCFIRST Reverse the order of writing the message and the CRC
- ADDDLENGTH Logically append the number of bytes in the record or file to the data being checksummed. Only the significant bytes of this count are included. The length is processed with the rightmost byte first.

Algorithmic operands are specified with 16-BIT or 32-BIT, one of which is required, followed by a word in hexadecimal that specifies the polynomial, followed by the remaining operands, which are all optional.

```

:
:      16-BIT      Use a 16-bit polynomial, which is specified in hexadecimal as the
:                  following word. Only the rightmost four digits are used.
:
:      32-BIT      Use a 32-bit polynomial, which is specified in hexadecimal as the
:                  following word.
:
:      ADDLENGTH   As described above. This is an alternative way to protect against a burst
:                  of leading zeros, as all CRC algorithms produce zero for input bytes of
:                  zero CRC when the accumulator is zero. That is, consider using
:                  ADDLENGTH when PRELOAD is zero.
:
:      COMPLEMENT  Equivalent to XOROUT FFFFFFFF.
:
:      PRELOAD     Specify the initial accumulator contents. The default is 0.
:
:      REFLIN      "Reflect input". Transpose the order of the bits in the input bytes so
:                  that the least significant bit is leftmost.
:
:      REFLOUT     "Reflect output". Transpose the order of the bits in each byte of the
:                  resulting CRC before applying XOROUT, but after applying ADDLENGTH.
:
:      XOROUT      Exclusive OR the CRC with the hexadecimal word that is specified after
:                  the keyword.

```

```

!      A number of built-in CRC algorithms may be selected, but they cannot be modified by
:      further options. Their parameters are listed in Figure 389 below.

```

Figure 389. Built-in CRC Algorithms

Name	Width	Poly	Initial	Xorout	R/i	R/o	AI
CRC-32	32-BIT	04C11DB7	FFFFFFFF	FFFFFFFF	YES	YES	NO
CRC-16	16-BIT	00008005	00000000	00000000	YES	YES	NO
CRC-16I	16-BIT	00008005	00008000	00000000	NO	NO	NO
CCITT-16	16-BIT	00001021	FFFFFFFF	FFFFFFFF	NO	NO	NO
CKSUM	32-BIT	04C11DB7	00000000	FFFFFFFF	NO	NO	YES

```

:      AUTODIN2 is a synonym for CRC-32 that hints at the heritage of the polynomial.

```

```

:      Operation: When a single output stream is defined, the default is to read the entire file,
:      compute the CRC, and write a single record at end-of-file. The record contains two, four,
:      or twelve bytes of binary data, depending on the parameters specified. When the
:      secondary output stream is defined, the input is passed to the primary output without being
:      delayed; the CRC is written to the secondary at end-of-file unless EACH is specified, in
:      which case a CRC is computed for each input record. When EACH is specified without
:      CRCFIRST, the primary output stream is written before the secondary output stream; the
:      order is reversed when CRCFIRST is specified.

```

```

!      A CRC record of twelve bytes is produced by the CKSUM algorithm, but only when using
!      the built-in version; specifying ADDLENGTH does not add this field. This record contains
:      four bytes CRC followed by eight bytes binary count of the number of bytes included in the
:      CRC.

```

```

:      Streams Used: Secondary streams may be defined. Records are read from the primary
:      input stream; no other input stream may be connected.

```


Record Delay: *crc* does not delay the record.

Commit Level: *crc* starts on commit level -2. It verifies that the primary input stream is the only connected input stream and then commits to level 0.

Premature Termination: *crc* terminates when it discovers that any of its output streams is not connected.

See Also: *digest*.

Examples:

```
pipe literal abc|crc cksum|spec 1-* c2x 1 | console
▶97B7E3490000000000000003
▶Ready;
pipe literal abc|xlate e2a|crc cksum|spec 1-* c2x 1 | console
▶48AA78A20000000000000003
▶Ready;
pipe literal abc|xlate e2a|crc cksum|spec 1.4 c2d 1 | console
▶ 1219131554
▶Ready;
```

On an ASCII Linux system:

```
j /home/john: echo -e -n abc|cksum
▶1219131554 3
```

With OMVS:

```
CCJOHN:/home/ccjohn: >echo -e -n abc|cksum
1073619496      10
```

The length 10 indicates that the flags are not respected and that a line end is appended (as it should be when -n is omitted).

Notes:

1. *crc* CKSUM interoperates with the POSIX *cksum* command, which produces the equivalent of

```
crc 32-bit 04c11db7 complement addlength
```

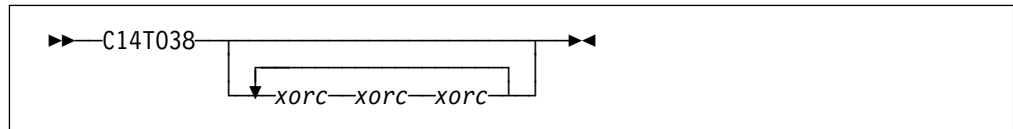
Note, however, that the UNIX command, when issued on an ASCII system, produces a different CRC than does *CMS Pipelines*, because the input data are not the same (X'414243' versus X'818283'). Also note that the POSIX *cksum* produces two unsigned decimal numbers (the second is the byte count) whereas *crc* in general produces a 2-byte or 4-byte binary number; *crc* CKSUM produces a 12-byte binary number.

Publications:

- Ross Williams: *A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS*.
- CRC32: *Ethernet specifications (Xerox, DEC, Intel)*, September 30 1980.
- CRC16: IBM form number GA22-6844-4 (IBM 2701 Data Adapter Unit OEM).
- CKSUM:
<http://www.opengroup.org/onlinepubs/009695399/utilities/cksum.html>

c14to38—Combine Overstruck Characters to Single Code Point

c14to38 combines overstruck lines of printer data replacing two overstruck characters with a single one. For example, a top left corner (⌈) overstruck on a top right corner (⌋) can be replaced with the “top T” (⌋⌈); an accent (˘) overstruck on a letter (e) can be replaced with an accented letter (è).



Type: Arcane filter.

Syntax Description: The argument string can have up to 255 conversion triplets. Each triplet consists of three blank-delimited words, each of which can be a single character or a two-character hexadecimal representation of a character. The default is to convert the 1403 box characters, generated by Document Composition Facility (SCRIPT/VS), to 3800 code points.

Operation: Input records are copied to the output until a record is met with a write no space operation code (X'01').

c14to38 tries to merge a record having write no space carriage control with the following record. Each position in the two records is inspected; if the characters are the first two of a triplet (in either order), a blank is stored in the first record and the third character of the triplet is stored in the second record. If the character in the first record is not blank and the character in the second record is blank, the two characters are swapped. The first record is discarded when it is blank from column 2 to the end; if not, it is written to the output. If the second record has X'01' carriage control, it is then merged with the next one, and so on.

Input Record Format: The first position of the record is a machine carriage control character.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *c14to38* delays or discards records that contain X'01' in column 1. It does not delay other records.

Premature Termination: *c14to38* terminates when it discovers that its output stream is not connected.

See Also: *optcdj* and *overstr*.

Examples: To print a document formatted for an IBM 1403 on an IBM 3800 printer or an all points addressable (APA) printer under control of Print Services Facility (PSF):

```
cp spool 00e fcb s8 char it12 ib12
cp tag dev 00e mvs system 0 OPTCD=J
pipe < $doc script | c14to38 | overstr | optcdj | printmc
cp close 00e
```

Notes:

1. The output is a 1403 type data stream as far as carriage control is concerned; no TRC is added.

dam—Pass Records Once Primed

dam waits for a record on its primary input stream. When it arrives, all streams are shorted to allow data to flow.



Type: Gateway.

Operation: *dam* waits for the first record to arrive on its primary input stream.

When the first record arrives on the primary input stream, *dam* shorts all input streams to the corresponding output stream (the dam bursts). The streams are shorted in numerical order. *dam* then terminates.

When end-of-file arrives on the primary input stream (that is, the primary input stream is empty), *dam* terminates without copying any records and without consuming any.

Streams Used: All streams are shorted if a record arrives on the primary input stream. No records are passed otherwise.

dam ignores end-of-file on its primary output stream; it propagates end-of-file between the two sides of streams 1 and higher.

Record Delay: *dam* strictly does not delay the record.

Commit Level: *dam* starts on commit level -2. It allocates the resources it needs and then commits to level 0.

Premature Termination: *dam* terminates when it discovers that no output stream is connected.

Converse Operation: *gate*.

See Also: *frtarget*, *predselect*, and *totarget*.

Examples:

To produce output only when the input contains a particular string:

dateconvert

```
/* Pass input to output only if target record found */
Parse Arg target /* String to be searched for. */
'CALLPIPE (endchar ?)',
  '? *:', /* Input from caller. */
  '| o: fanout', /* Divert copy for output. */
  '| locate' target, /* Select target record(s). */
  '| d: dam', /* First one opens floodgate. */
  '? o:', /* The input records */
  '| elastic', /* Hold until dam bursts. */
  '| d:', /* Possibly go over the dam. */
  '| *:' /* Output to caller. */
```

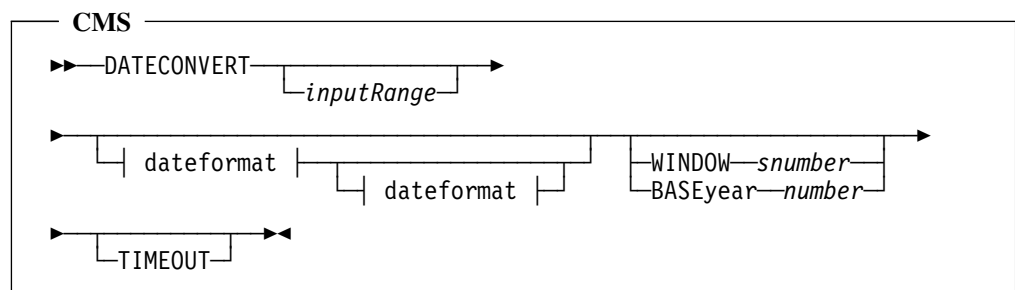
Notes:

1. Though *dam* does not delay the record, it does not produce output until it has sensed a record on its primary input stream. You are likely to need something upstream on the secondary input stream to hold the records while the decision is being made about processing the records.

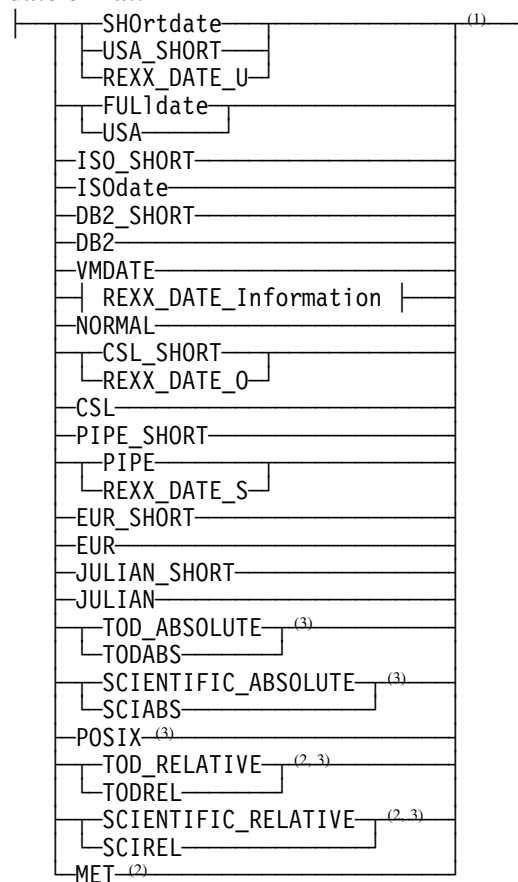
dateconvert—Convert Date Formats

dateconvert changes the contents of a single date field from one format to another.

This article is a synopsis; refer to the description of the `DateTimeSubtract` service in *z/VM: CMS Callable Services Reference*, SC24-6259, for further information.



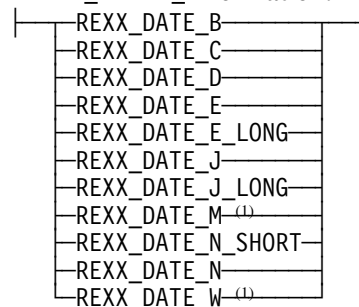
dateformat:



Notes:

- ¹ The formats grouped together are synonyms for each other; they have the same format definition. The REXX_DATE_x formats can also be specified as REXXx or Rx. The x can be specified as B, C, D, E, E_LONG, J, J_LONG, M, N, N_SHORT, O, S, U or W.
- ² This is a relative format; the rest of the formats are absolute. You cannot convert between relative and absolute formats.
- ³ Time is an integral part of these formats; therefore, the TIMEOUT operand is ignored for these formats when they are used as an output format.

REXX_DATE_Information:



Note:

- ¹ This format is valid only for the output format parameter.

dateconvert

Type: Filter.

Syntax Description: An optional input range specifies the date field to be converted; the default input range is the entire record. The input date format and output date format may be specified. The default input date format is SHORTDATE; the default output date format is ISODATE.

Either a sliding WINDOW or fixed BASEYEAR may be specified to compute the century of the output date format. The default is to use a sliding window that begins fifty years in the past and extends forty-nine years into the future.

When TIMEOUT is specified the time component of the output format is included in the output; the default is to suppress the time component. When the input field also contains a time value, the value is included in the transformation; the default is to use midnight of the specified date.

The date formats supported by *dateconvert* are listed in Figure 390 on page 356 and Figure 391 on page 359.

Operation: If the specified input range is not present in the record, the record is passed unchanged to the primary output stream and no further action is taken.

If the input range is present in the input record, the contents of the specified range are converted as requested.

When the conversion succeeds, the conversion result replaces the specified range in the record and the updated record is written to the primary output stream. Note that the output record can have a different length than the input record.

When the conversion fails and the secondary output stream is defined, the input record is passed to the secondary output stream.

dateconvert terminates with an error message when the conversion fails and the secondary output stream is not defined.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *dateconvert* strictly does not delay the record.

Commit Level: *dateconvert* starts on commit level -2. It verifies its arguments and then commits to level 0.

Premature Termination: *dateconvert* terminates when it discovers that any of its output streams is not connected.

See Also: *greg2sec* and *sec2greg*.

Examples:

```
! pipe cp query time | take 1 | cons | dateconvert w-1 vmdate rexxj | ...
!                                     ... console
!
! ▶TIME IS 14:50:30 EDT WEDNESDAY 04/29/20
! ▶TIME IS 14:50:30 EDT WEDNESDAY 20120
! ▶Ready;
```

```

    pipe literal 03/09/46 | dateconvert | console
▶2046-03-09
▶Ready;

    pipe literal 03/09/46 | dateconvert window -99 | console
▶1946-03-09
▶Ready;

    pipe literal 03/09/46 | dateconvert baseyear 1925 | console
▶1946-03-09
▶Ready;

    pipe literal 03/09/46 | dateconvert baseyear 1130 | console
▶1146-03-09
▶Ready;

    pipe literal 02/29/46 | dateconvert | cons
▶FPLRIC1183E Date cannot be converted; input date 02/29/46 is not valid
▶FPLMSG003I ... Issued from stage 2 of pipeline 1
▶FPLMSG001I ... Running "dateconvert"
▶Ready(01183);

    pipe literal 03/09/46 | dateconvert short rexxn window -60 | console
▶9 Mar 2046
▶Ready;

```

Date Formats

Figure 390 on page 356 and Figure 391 on page 359 describe the input format and output format parameters for absolute formats and relative formats respectively. The formats grouped together are synonyms.

The symbols used in the definition column of these tables have the following meanings:

Date Symbols

mm specifies the 2-digit month.

mmm specifies the three-character English name of the month:

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

mmmmmmmmmm specifies the English name of the month:

January February March April May June July August September October
November December

dd specifies the 2-digit day of the month.

ddd specifies the day number in the year.

dddd specifies the 1 to 5-digit REXX Century date.

dddddddddd specifies the 6 to 10-digit REXX Base date.

wwwwwwwww specifies the English name of the day of the week:

Sunday Monday Tuesday Wednesday Thursday Friday Saturday

yy specifies the 2-digit year.

yyyy specifies the 4-digit year.

yyyyyyy specifies the 1 to 7-digit year.

Time Symbols

dateconvert

- ! *hh* specifies the 2-digit hours.
- ! *mm* specifies the 2-digit minutes.
- ! *ss* specifies the 2-digit seconds.
- ! *uuuuuu* specifies millionths of a second.

! *Figure 390 (Page 1 of 4). Absolute input format and output format Parameters*

Format	Definition	Length	Output Left or Right Justified
SHOrtdate USA_SHORT REXX_DATE_U REXXU RU	<i>mm/dd/yy hh:mm:ss.uuuuuu</i> specifies the REXX USA date. This is the default for the input format parameter.	8 or 24	Left
FULLDATE USA	<i>mm/dd/yyyyyy hh:mm:ss.uuuuuu</i> specifies the FULLDATE date.	7-29	Left
ISO_SHORT	<i>yy-mm-dd hh:mm:ss.uuuuuu</i> specifies the ISO_SHORT date.	8 or 24	Left
ISODate	<i>yyyyyy-mm-dd hh:mm:ss.uuuuuu</i> specifies the ISODATE date. This is the default for the output format parameter.	7-29	Left
DB2_SHORT	<i>yy-mm-dd-hh.mm.ss.uuuuuu</i> specifies the DB2_SHORT date.	8 or 24	Left
DB2	<i>yyyyyy-mm-dd-hh.mm.ss.uuuuuu</i> specifies the DB2 date. This is the default for the output format parameter.	7-29	Left
VMDATE	<p>User's virtual machine date format setting. Issue the CP QUERY DATEFORMAT command to query the date format setting, and then find the corresponding format in the table.</p> <p>It can be set on a system-wide basis and also for the individual user. The system-wide default date format is set with the SYSTEM_DATEFORMAT system configuration statement. The user's default date format is set with the DATEFORMAT user directory control statement. The system-wide default and the user's default can also be set with the CP SET DATEFORMAT command. The user's default date format is set to the system-wide default.</p> <p>The hierarchy of possible date format settings from highest priority to lowest, is:</p> <ul style="list-style-type: none"> • User default • System-wide default 		

Figure 390 (Page 2 of 4). Absolute input format and output format Parameters

Format	Definition	Length	Output Left or Right Justified
REXX_DATE_B REXXB RB	<p><i>ddddddddd hh:mm:ss.uuuuuu</i> specifies the REXX Base date.</p> <p>The number of complete days (that is, not including the current day) since and including the base date, 1 January 0001.</p> <p>Note: The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). The REXX_DATE_B does not consider any errors in the calendar system that created the Gregorian calendar originally.</p> <p>For input, you may supply up to 10 day digits, provided the number of days represented can be expressed without overflow in a signed fullword. The output date is not padded with leading zeroes.</p>	10 or 26	Right
REXX_DATE_C REXXC RC	<p><i>dddd/hh:mm:ss.uuuuuu</i> specifies the REXX Century date.</p> <p>For input, you may supply up to 5 day digits.</p> <p>The number of days, including the current day, since and including January 1 of the last year that is a multiple of 100. The REXX_DATE_C is never negative. When <i>dateconvert</i> returns the date in this format, it never pads the day field with leading zeroes.</p> <p>A sliding window has no effect on this date format. If specifying a fixed window with BASEYEAR, <i>dateconvert</i> ignores the last two digits of the year. For example, if you specify BASEYEAR 1947, <i>dateconvert</i> uses 1900 for the base year.</p>	5 or 21	Right
REXX_DATE_D REXXD RD	<p><i>ddd hh:mm:ss.uuuuuu</i> specifies the REXX Days date.</p> <p>The <i>ddd</i> has no leading zeroes. On input, the <i>ddd</i> is interpreted as being that day of the base year. On output, the <i>ddd</i> is returned as the day number within the input year.</p> <p>A sliding window has no effect on this date format.</p>	3 or 19	Right
REXX_DATE_E REXXE RE	<p><i>dd/mm/yy hh:mm:ss.uuuuuu</i> specifies the REXX European date.</p>	8 or 24	Left
REXX_DATE_E_LONG REXXE_LONG RE_LONG	<p><i>dd/mm/yyyyyy hh:mm:ss.uuuuuu</i> specifies the REXX European Date Long.</p>	7-29	Left
REXX_DATE_J REXXJ RJ	<p><i>yyddd hh:mm:ss.uuuuuu</i> specifies the REXX Julian date.</p>	5 or 21	Left

dateconvert

! *Figure 390 (Page 3 of 4). Absolute input format and output format Parameters*

Format	Definition	Length	Output Left or Right Justified
REXX_DATE_J_LONG REXXJ_LONG RJ_LONG	yyyyddd hh:mm:ss.uuuuuu specifies the REXX Julian Date Long.	7 or 23	Left
REXX_DATE_M REXXM RM	mmmmmmmmmm specifies the REXX Month date. This is valid only for the output format parameter.	9	Left
REXX_DATE_N_SHORT REXXN_SHORT RN_SHORT	dd mmm yy hh:mm:ss.uuuuuu specifies the REXX Normal Date Short. The month must be specified in English and case is significant.	9 or 25	Right
REXX_DATE_N REXXN RN	dd mmm yyyy hh:mm:ss.uuuuuu specifies the REXX Normal date.	11 or 27	Right
NORMAL	dd mmm yyyy hh:mm:ss.uuuuuu specifies the Normal date. On input, the month must be specified in English and the case is not significant.	8-30	Left
REXX_DATE_O REXXO RO CSL_SHORT	yy/mm/dd hh:mm:ss.uuuuuu specifies the REXX Ordered date.	8 or 24	Left
CSL	yyyyyy/mm/dd hh:mm:ss.uuuuuu specifies the CSL date.	7-29	Left
PIPE_SHORT	yyymmddhhmmssuuuuuu specifies the PIPE_SHORT date.	6 or 18	Left
REXX_DATE_S REXXS RS PIPE	yyymmddhhmmssuuuuuu specifies the REXX Standard date.	8 or 20	Left
REXX_DATE_W REXXW RW	wwwwwwwww specifies the REXX Weekday date. This is valid only for the output format parameter.	9	Left
EUR_SHORT	dd.mm.yy hh:mm:ss.uuuuuu specifies the EUR_SHORT date.	8 or 24	Left
EUR	dd.mm.yyyyyy hh:mm:ss.uuuuuu specifies the EUR date.	7-29	Left
JULIAN_SHORT	yy.ddd hh:mm:ss.uuuuuu specifies the JULIAN_SHORT date.	6 or 22	Left
JULIAN	yyyyyy.ddd hh:mm:ss.uuuuuu specifies the JULIAN date.	5-27	Left

! Figure 390 (Page 4 of 4). Absolute input format and output format Parameters

! Format	! Definition	! Length	! Output ! Left or ! Right ! Justified
! TOD_ABSOLUTE ! TODABS	! Unsigned doubleword indicating the number of TOD ! clock units that have elapsed between the standard ! epoch and the moment being expressed. ! A <i>TOD clock unit</i> is a unit of time, just as minutes, ! seconds, and hours are units of time; there are 4096 ! TOD clock units in a microsecond. ! The standard epoch is the moment at which the TOD ! clock would have read X'00000000 00000000'. On a ! virtual machine system the standard epoch is January ! 1, AD 1900, 00:00:00 UTC. This format is exactly ! the format of the output of the STORE CLOCK (STCK) ! instruction.	! 8	! Left
! SCIENTIFIC_ABSOLUTE ! SCIABS	! Eight-byte value containing two signed four-byte ! binary integers. ! The first integer is the number of whole days (that is, ! 24-hour units) that have elapsed between January 1, ! 4713 BC, Noon, Universal Time, Coordinated (UTC) ! and the moment being expressed. This is called the ! <i>Julian day number</i> . ! The second integer is the number of milliseconds to be ! added to the Julian day number to reach the moment ! being expressed. If the input is date specific, the time ! will be zero. This results in the time output portion to ! always be Noon. ! The first integer must be greater than or equal to zero. ! The second integer must be greater than or equal to ! zero and less than the number of milliseconds in a day ! (86400000).	! 8	! Left
! POSIX	! Unsigned doubleword indicating the number of ! seconds since the POSIX epoch (this is defined ! according to IEEE standard 1003.1). The POSIX ! epoch is January 1, AD 1970, 00:00:00 UTC.	! 8	! Left

! Figure 391 (Page 1 of 2). Relative input format and output format Parameters

! Format	! Definition	! Length	! Output ! Left or ! Right ! Justified
! TOD_RELATIVE ! TODREL	! Signed doubleword specifying an <i>amount</i> of time ! rather than a specific date. Specified in TOD clock ! units.	! 8	! Left

deal

! Figure 391 (Page 2 of 2). Relative input format and output format Parameters			
! Format	! Definition	! Length	! Output Left or Right Justified
! SCIENTIFIC_RELATIVE SCIREL	! Eight-byte value containing two signed four-byte binary integers and specifying an <i>amount</i> of time rather than a specific date. ! The first integer is a number of whole days (that is, 24-hour units). The second integer represents a fractional day and is the number of milliseconds to be added to the whole day count. ! There are no constraints on the value of the first integer. The second integer must be greater than or equal to zero and less than the number of milliseconds in a day (86400000).	! 8	! Left
! MET	! [-]ddddddddd hh:mm:ss.uuuuuu specifies the MET format. ! For input, you may supply up to 10 day digits, provided the number of days represented can be expressed without overflow in a signed fullword. When <i>dateconvert</i> returns the date in MET format, it never pads the day field with leading zeroes. This format, <i>Mission Elapsed Time</i> , is used by NASA for measuring the duration of space flights.	! 11-27	! Right

Notes:

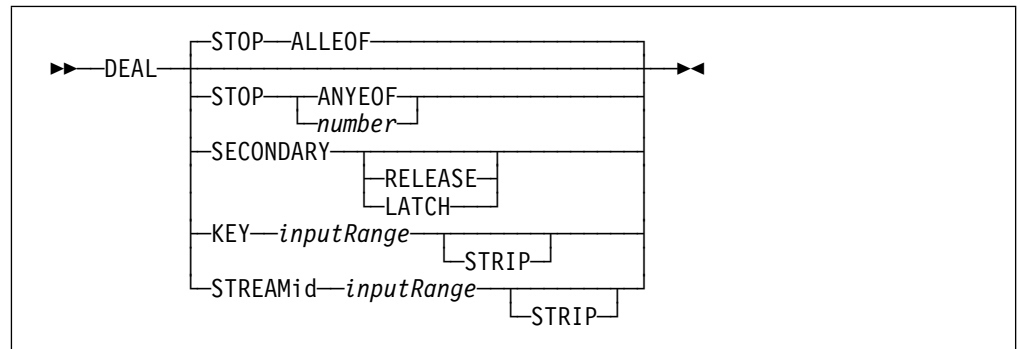
1. Leap seconds are ignored in the calculations.
2. *dateconvert* assumes that the first day of the Gregorian calendar is September 14, 1752. The day before that is September 2, 1752; it is in the Julian calendar.
3. Unlike the REXX `date()` function, the first date format in *dateconvert* specifies the input format and the second date format specifies the output format.
4. *dateconvert* does not provide a way specify the separator characters for input and output date format. A *change* or *xlite* stage before or after *dateconvert* can be used to transform the date into the required format.
5. Transformation between TOD and ISO date and time format can also be done in *spec* using the C2T and T2C conversion which also supports a time zone offset. Because this transformation does not use the `DateTimeSubtract` service, the use of *spec* may offer some performance advantages.

***deal*—Pass Input Records to Output Streams Round Robin**

deal reads records from its primary input stream and passes each record to one of its output streams. Thus, *deal* can be used to divide work between a number of “worker stages”.

By default, records are written to the output streams round robin, that is, the first record to the primary output stream, the second record to the secondary output stream, and so on.

Alternatively, you can supply the stream identifier for the stream to receive a record or you can specify that a run of records containing the same key are written to the same output stream.



Type: Gateway.

Syntax Description: Arguments are optional; the default is STOP ALLEOF. The four options are exclusive; specify at most one of them.

STOP	Specify the condition under which <i>deal</i> should terminate prematurely. ALLEOF, the default, specifies that <i>deal</i> should continue as long as at least one output stream is connected. ANYEOF specifies that <i>deal</i> should stop as soon as it determines that an output stream is no longer connected. A number specifies the number of unconnected streams that will cause <i>deal</i> to terminate. The number 1 is equivalent to ANYEOF.
SECONDARY	The secondary input stream contains the stream identifiers of the streams that are to receive the records on the primary input stream.
RELEASE	Consume the record on the secondary input stream before reading the record from the primary input stream.
LATCH	<i>deal</i> waits for a record to arrive on its input streams. When a record arrives at the secondary input stream, the specified output stream is selected and the record is discarded; that is, LATCH implies RELEASE. When a record arrives at the primary input, it is copied to the currently selected output. The primary output stream is selected initially.
KEY	Specify the input range that contains the key of each record. Runs of records that contain the same key are written to the same output stream.
STREAMID	Specify the input range within the record on the primary input stream that contains the stream identifier to receive the record. Each record is routed individually.
STRIP	Delete the key or stream identifier from the output record. The <i>inputRange</i> must be either at the beginning or at the end of the record.

Operation: An input record is passed to one of the output streams.

When KEY, SECONDARY, and STREAMID are omitted, records are passed to the output streams round robin. The first input record is passed to the primary output stream, the second input record is passed to the secondary output stream, and so on until it wraps after the highest-numbered output stream. The next record is then passed to the primary output stream and the cycle is repeated. *deal* reacts to end-of-file on an output stream by trying

deal

the next output stream until as many streams are at end-of-file as specified in the STOP option.

When KEY is specified, the first record is passed to the primary output stream and the contents of its key field are then stored in a buffer. A run of records that contain the stored key is passed to the same output stream. When the key of the input record is not equal to the stored key, the next output stream is selected and the run of records is passed to this stream.

When SECONDARY is specified and LATCH is omitted, a pair of records from each input stream is processed together. *deal* first peeks at the record on the secondary input stream to determine where to write the record on the primary input stream. The record from the secondary input stream specifies the number or identifier of the stream to receive the corresponding record from the primary input stream. The record from the primary input stream is then passed to the specified stream. When RELEASE is specified, the record on the secondary input stream is consumed before the primary input stream is read; otherwise, the two records are consumed, the one on the primary input stream first, after the output record has been written.

When SECONDARY LATCH is specified, records on the secondary input stream specify the output stream to be selected; such records are discarded immediately. Subsequent records on the primary input stream are passed to the stream last selected by a record on the secondary input stream.

When STREAMID is specified, the record is routed based on its contents. The contents of the specified range determine the output stream to receive the record.

Input Record Format: When SECONDARY is specified, the secondary input stream contains one word per record. This word is the stream identifier for the output stream that should receive the corresponding record from the primary input stream. The stream identifier can be a number, in which case it is the number of the stream (from 0); or it can be an alphanumeric stream identifier.

Streams Used: Records are read from the primary input stream. If SECONDARY is specified, records are also read from the secondary input stream; the two input streams are synchronised unless RELEASE is specified. Records are, in general, written to all connected output streams.

Record Delay: *deal* strictly does not delay the record.

Commit Level: *deal* starts on commit level -2. It verifies that only the primary input stream is connected unless SECONDARY is specified, in which case the secondary input stream must be connected; and then commits to level 0.

Premature Termination: *deal* terminates as soon as it receives end-of-file on any of its input streams.

When KEY, SECONDARY, or STREAMID is specified, *deal* terminates as soon as it discovers that an output stream is at end-of-file; the corresponding input records are not consumed.

When STOP is specified or defaulted, it specifies how many output streams can go to end-of-file before *deal* terminates. The corresponding input records are not consumed.

Converse Operation: *gather*.

See Also: *fanout*.

Examples: To pass an input record to one of three virtual machines, as a special message; that is, to spread the load among multiple servers:

```
'callpipe (end ? name DEAL.STAGE:120)',
  '?*:',
  '|d:deal',
  '|change //MSG BEE1 /',
  '|cp',
  '?d:',
  '|change //MSG BEE2 /',
  '|cp',
  '?d:',
  '|change //MSG BEE3 /',
  '|cp'
```

To discard every second record:

```
'callpipe (end ? name DEAL.STAGE:151)',
  '?*:',
  '|d:deal',
  '|*:',
  '?d:',
  '|hole'
```

Note that the records are discarded by *hole*. Leaving an output stream unconnected will not cause records to be dropped; *deal* will try to write the record until it succeeds or until it has received sufficient end-of-file indications to terminate.

To hand out work to one of several servers:

```
/* Passes an input record to whichever output is ready.          */
Signal on novalue

/*****
/* This subroutine distributes work amongst server pipelines, which */
/* are connected from output 0 to input 1, from output 1 to input 2, */
/* and so on (this makes for easy coding of the main pipeline).    */
/*                                                                    */
/* A server writes a record (a "ready" token) when it is ready to */
/* process a request. An input record will then be made available. */
/* The server should consume this input record "quickly" (to avoid */
/* blockage of the DEAL stage that doles out work to the performing */
/* pipelines).                                                       */
/*                                                                    */
/* Each server pipeline should contain a pipeline that produces */
/* ready tokens as appropriate. For example, you can use a sipping */
/* pipeline along these lines:                                       */
/*                                                                    */
/*     output ready!                                                */
/*     callpipe *|take 1|<server>                                     */
/*                                                                    */
/* Each "ready" token is then turned into the stream's number and */
/* fed into the buffer of available streams.                         */
*****/

'maxstream output'
maxstream=RC-1
```

deal

```
If RC=0
  Then exit 999

pipe=''

first='tod:|'
do j=0 to maxstream
  pipe=pipe,
    '\*..'j+1':',
    '|spec /'j'/ 1',
    '|i:',
    '\first'd:',
    '\*..'j+1':',
  first=''
end

'callpipe (end \ name THROTTLE.REXX:31)',
  '|i:faninany',          /* Ready records      */
  '|elastic',            /* Form a queue       */
  '|tod: fanout',        /* Couple forward     */
  '\*..0:',
  '|d:deal secondary',   /* Distribute         */
  '\*..0:',              /* to server thread   */
  pipe

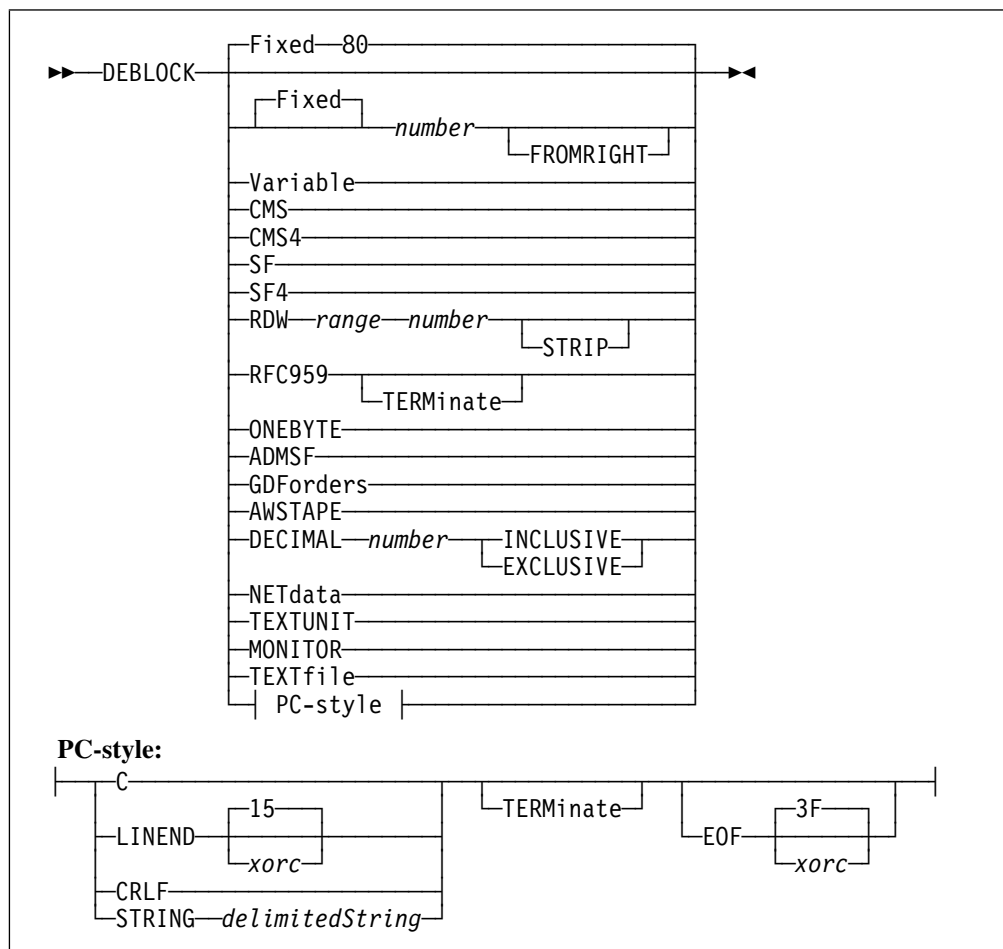
exit RC
```

Notes:

- : 1. It is unlikely to be efficient to distribute work round robin amongst identical processing stages that do not in some way connect to a different task or another virtual machine.
- : 2. You should ensure that no two records arrive concurrently when SECONDARY LATCH is specified; the order of processing is unspecified and in general random if they do.
- :

deblock—Deblock External Data Formats

deblock generates output *records* from input *blocks*. Input blocks are often received from other systems, for example, a tape in VBS format from a z/OS system or a binary file uploaded from a workstation.



Type: Filter.

Syntax Description: The argument is optional. If present, it must be a keyword or a number. `FIXED` is the default; you must write a number after `FIXED` when it is used explicitly. The default is to deblock to 80-byte records.

Operation: In general, *deblock* performs the inverse operation of *block*. Input record boundaries are ignored in some input formats, because the file itself contains the information required to reconstruct the logical record structure; such input files are called *byte streams*.

deblock

FIXED	<p>Produce as many records of fixed length as required for each input block. Output records do not span input blocks; a short record is written for the last part of the block if a block is not an integral multiple of the record length. Thus, <i>deblock</i> FIXED accepts input records that cannot be created with <i>block</i> FIXED.</p> <p>Normally, <i>deblock</i> FIXED processes the input block from left to right, but the order is reversed when FROMRIGHT is specified. The first part of the block is then written last; it may be a short record.</p>
VARIABLE	<p>Deblock OS variable length records. <i>deblock</i> v supports all four OS variable formats: v, VB, VS, and VBS. Null segments are discarded. Extended block descriptor words are supported.</p>
CMS	<p>Reconstruct a file from the format used to store variable record format files internally in the CMS file system. Internally, each logical record has a halfword prefix containing the length of the record. Length zero means end-of-file. The length field is not present in the output record. The input is considered a byte stream.</p>
CMS4	<p>Reconstruct a file where each logical record is prefixed four bytes length that specifies the number of data characters that follow. Length zero means end-of-file. The length field is not present in the output record. The input is considered a byte stream.</p>
SF	<p>Deblock structured fields. A structured field consists of a halfword (sixteen bits) length field and a variable length data field. The contents of the length field includes those two bytes. Thus, a null structured field consists of the data X'0002'. The halfword length field is not present in the output record. The input is considered a byte stream; the structured fields can be spanned over input records.</p>
SF4	<p>Logical records are prefixed four bytes length. The length field contains four plus the count of data characters that follow. That is, the length field specifies the length of the logical record inclusive of the record descriptor. Length zero means end-of-file. The length field is not present in the output record. The input is considered a byte stream.</p>
RDW	<p>Logical records contain a descriptor word in the positions defined by <i>range</i>. The contents are treated as a binary unsigned number. The <i>number</i> specifies an overhead to be added to the contents of this record descriptor word to determine the length of the logical record, inclusive of the descriptor word. STRIP specifies that the record descriptor word is not written as part of the output record; the range must then begin in column 1.</p>
RFC959	<p>Deblock according to the format defined in Request For Comments 959 ("File Transfer Protocol"). A convenience for rdw 2.2 3. Specify TERMINATE to have <i>deblock</i> terminate without issuing an error message when the last record is incomplete (which may happen when an ABOR command is issued to the FTP Server).</p>
ONEBYTE	<p>Deblock logical records that consist of a one byte length field and a variable length data field. The contents of the length field includes this one byte. Thus, a null one byte record consists of the data X'01'. The length field is not present in the output record. The input is considered a byte stream; the logical record can be spanned over input records.</p>

ADMSF	Deblock GDDM structured fields. When a logical record ends one character before the end of a block and the remaining character is X'00', this pad character is ignored. The halfword length field is not present in the output record.
GDFORDERS	Deblock orders in an already deblocked GDF structured field.
AWSTAPE	Reconstruct tape blocks from an emulated tape. The input is considered a byte stream.
DECIMAL	Deblock records prefixed by their length in decimal. The number specifies the width of the sequence field, which contains normal printable digits. Specify INCLUSIVE if the length of the count field is included in the count; EXCLUSIVE if not. The output record contains the count field at the beginning. The input is considered a byte stream.
NETDATA	Records in netdata format are deblocked and built from segments such that the first character of each output record is the flag byte, which always has the leftmost two bits (X'C0') on indicating a complete record. It is verified that all input records except the last one have the same length. This saves you from subtle errors when a truncated record is let in from SPOOL. The input is considered a byte stream, but the remainder of the input record is discarded after an \INMR6 record is processed. This is done to support stacked reader files. Refer to "Netdata Format" on page 65 for usage information.
TEXTUNIT	Write a record for each text unit in netdata control records. Input records must be control records with X'E0' in the first position; columns 2-6 must contain the literal 'INMR0'; column 7 must be 1, 2, 3, 4, 6, or 7. The file count in record type INMR02 is discarded.
MONITOR	Deblock VM Monitor blocks. Each record is prefixed a halfword length field as for SF. Records are not spanned across blocks. Blocks are padded with binary zeros, which are discarded.
TEXTFILE	A convenience for LINEND 15 TERMINATE. The input file need not be in the format produced by <i>block</i> TEXTFILE. The input is considered a byte stream.
C LINEND	Two keywords support deblocking logical records that are separated by a line end character in the input block. The line end character is removed from the logical records when they are written to the output stream. The input is considered a byte stream; logical records can span input blocks in these formats. In format C, logical records are separated by a newline character, X'25' (line feed). <i>deblock</i> LINEND is similar; the default line end character is X'15'. You can specify the line end character as a single character or a two-character hex code after the keyword LINEND. A null record is written for two consecutive end of line characters.
CRLF	Deblock lines separated by carriage return line feed (X'0D25'). Logical records can span blocks, as can the line end sequence. A null record is written for two consecutive carriage return line feed sequences. The input is considered a byte stream.
STRING	Deblocks lines separated by the specified string. Logical records can span blocks, as can the separator string. A null record is written for two consecutive occurrences of the specified string. The input is considered a byte stream.

deblock

The formats C, LINEND, CRLF, and STRING support two keywords, TERMINATE and EOF. Specify TERMINATE to suppress a trailing null line when the file ends in a line end sequence. Use EOF to specify an end-of-file character. *deblock* discards the end-of-file character and all remaining input. That is, *deblock* consumes the first record that contains an end-of-file character and then terminates. X'3F' (substitute) is the default end-of-file character. *deblock* scans for an end-of-file character only when the keyword is specified.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *deblock* delays input records as required to build an output record. The delay is unspecified.

Premature Termination: *deblock* terminates when it discovers that its output stream is not connected. *deblock* TEXTUNIT terminates when it has consumed the input record that contains the INMR06 control record that terminates the file.

Converse Operation: *block*.

See Also: *fblock* and *spill*.

Examples: FMTPCBIN REXX reformats a text file uploaded from a PC as a binary:

```
/* Format PC binary.                                     */
signal on novalue
'callpipe (name FMTPCBIN)',
  |*:',
  |xlate a2e',          /* From ASCII to EBCDIC      */
  |deblock c terminate eof', /* Deblock from newlines  */
  |strip trailing 0d 1',   /* Discard one trailing line feed */
  |*:'
exit RC
```

The file AUTOEXEC BATBIN is uploaded in binary:

```
pipe < autoexec batbin | fmpcbin | take 3 | console
►ECHO OFF
►SET INDLANG=DK
►INDKB
►Ready;
```

To deblock a file with X'15' line end characters:

```
...| deblock linend |...
```

To extract the text units describing the data set from a reader file in the NETDATA format:

```
/* Get text units from reader file */
'PIPE',
  | reader',
  | find' '41'x,
  | spec 2-* 1.80',
  | deblock net',
  | find' 'e0'x,
  | deblock textunit',
  | ...
```

To deblock an ADMGDF file:

```

/* Deblock ADMGDF */
'PIPE (name DEBLOCK)',
  '|< x admgdf',          /* Read gdf file          */
  '|drop 1',             /* Drop descriptor        */
  '|spec 21-*',          /* Drop record identifier */
  '|deblock admsf',      /* Unravel GDDM structured fields */
  '|deblock gdf',        /* Now unblock the orders */
  '|...'

```

To restore the record format of a LIST3820 (or similar) file that has lost its record boundaries during file transfer:

```
... | deblock rdw 2.2 1 | ...
```

The records contain a carriage control character (one byte), which is followed by a structured field. The first two bytes of the structured field contain the length of the field, inclusive of these two bytes, but it does not include the carriage control character; therefore, the adjustment factor is one.

Notes:

1. VB, VS, and VBS are synonyms for VARIABLE.
2. Though input records are called blocks and output records are called logical records, these are still records (or lines) as perceived by the pipeline dispatcher.
3. Netdata and PC deblocking may produce output records that contain data from several input records; in this respect, *deblock* can be considered to be blocking input records rather than deblocking them.

delay—Suspend Stream

delay copies an input record to the output at a particular time of day or after a specified interval has elapsed. The first word of each input record specifies the time at which it is to be passed to the output.

►►—DELAY—◄◄

Type: Device driver.

Placement: *delay* must not be a first stage.

Operation: The input record is copied to the primary output stream when the delay expires (the specified time is reached or the specified interval elapses). A record that specifies a time of day (that is, without a leading plus) is copied to the output immediately when it is read after the time specified.

Input Record Format: The first word of each input record specifies when the record is to be copied to the output; the remainder of the record is not inspected.

The delay is a word that contains up to three numbers separated by colons. A leading plus indicates that the time is relative to the time of day when the record is read; with no leading plus, the time is local time relative to the previous midnight.

The numbers represent hours, minutes, and seconds. The seconds field may contain a decimal point and up to six fractional digits, giving microsecond resolution. The numbers

delay

must be zero or positive, but are not restricted to the normal conventions for seconds per minute, minutes per hour, and hours per day. You can wait until 1:17:64, which is equivalent to 1:18:04. You can wait until any time in the future, as long as the time-of-day clock has not changed sign. (If your system is using the standard epoch, the sign will change in 2041.) A delay of 8760 waits until midnight on the 365th day following the current day. (Assuming the system stays up that long and assuming no drift of the time-of-day clock.)

When the first word has one or two numbers, the interpretation depends on the presence of a leading plus. With the plus (indicating a relative interval), the rightmost number is taken to be seconds; a further number to the left of it represents minutes. When there is no leading plus (a time of day is specified), the leftmost number represents the hour; minutes and seconds are assumed to be zero when not specified.

Streams Used: *delay* passes the input to the output.

Record Delay: *delay* strictly does not delay the record. That is, *delay* consumes the input record after it has copied it to the primary output stream; records are delayed *in time*, but the relative order of records that originate in a particular filter is unchanged.

Premature Termination: *delay* terminates when it discovers that its output stream is not connected; *delay* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

Examples: To perform an action at 3 am every day. Note the code to determine whether the subroutine pipeline was called before or after 3 am.

```
/* 3AM REXX */
If time('Hours') > 2      /* 3am or later?          */
  Then addl=''
  Else addl='literal 3|' /* No, wait till then          */
'PIPE',
  'literal 27|',         /* 3am tomorrow                */
  'dup *|',             /* Forever                      */
  addl,                 /* Maybe 3am today?            */
  'delay|',             /* Wait                          */
  '*:'
```

To issue a CP command once a minute:

```
pipe literal +60 | dup * | delay | spec ,QUERY RDR ALL, | cp |...
```

To issue the command “midnight” at midnight:

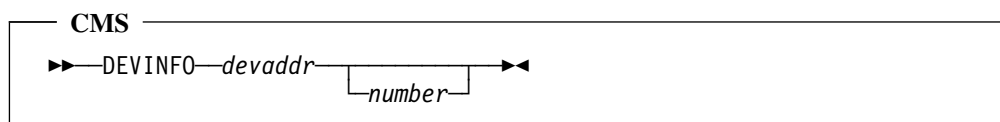
```
pipe literal 24 | delay | spec ,midnight, | subcom cms
```

Notes:

1. On CMS, *delay* issues diagnose 0 to determine the time zone offset.
2. `literal 0 | delay` never waits; use 24 to wait until the next midnight.
3. *delay* does not depend on the date of the *epoch* (the date corresponding to a zero value of the time-of-day clock). The epoch must begin at midnight GMT when waiting to a particular time of day.
4. No more than 16 *delay* stages can be active concurrently on z/OS.

devinfo—Write Device Information

devinfo writes information about a range of virtual devices. The output includes at least the device number and its generic type. When further information is available, it is included in the output record.



Type: Device driver.

Placement: *devinfo* must be a first stage.

Syntax Description:

devaddr The first or only device number to query.

number The count of devices to query. The default count is zero.

Output Record Format: For each device, the output line contains as a minimum the device number and its generic type.

For devices that respond to E4 sense, the next two words contain device type and control unit type.

For a CKD device, the rest of the line contains the number of primary tracks, number of tracks per cylinder, and usable space per track.

For FBA, the rest of the line contains block size, blocks per track, blocks per cylinder, blocks under movable heads, and blocks under fixed heads.

Premature Termination: *devinfo* terminates when it discovers that its output stream is not connected.

Examples:

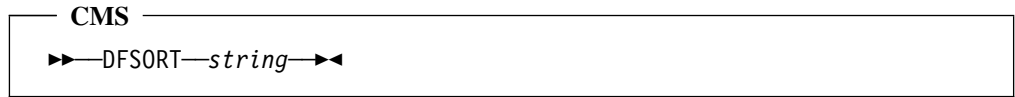
```

pipe devinfo 9|cons
▶0009 TERM
▶R;
pipe devinfo 180|cons
▶0180 TAPE 3480 3480
▶R;
pipe devinfo 190|cons
▶0190 DASD 3390 3990 107 15 58786
▶R;
pipe devinfo 100|cons
▶0100 FBA 9336 6310 512 111 777 4000 0
▶R;

```

dfsort—Interface to DFSORT/CMS

dfsort builds a parameter list to call DFSORT/CMS, inserts the input file on the E15 exit, and extracts the sorted file from the E35 exit and writes it into the pipeline.



Type: Sorter.

Placement: *dfsort* must not be a first stage.

Syntax Description: The arguments are optional. Specify sort control statements in the parameter string. The string passed to the sort is made upper case. *dfsort* adds this statement to the end of the specified string:

```
RECORD TYPE=V,LENGTH=32760
```

Input records are always variable record format as far as DFSORT/CMS is concerned; *dfsort* adds and removes record descriptor words. Because the record descriptor word occupies the first four positions of the record as seen by DFSORT/CMS, the sort fields **must** specify a field position that is four larger than the position in the record that is passed to *dfsort*.

Input Record Format: Records can be up to 32756 bytes long.

Record Delay: *dfsort* delays all records until end-of-file.

Premature Termination: *dfsort* terminates when it discovers that its output stream is not connected.

See Also: *collate*, *merge*, and *sort*.

Examples: To sort on columns 1 to 5 of the input record:

```
... | dfsort option nolist sort fields=(5,5,ch,a) | ...
```

Note that you must add four to the column number because the input records that DFSORT/CMS sees are prefixed four bytes record descriptor.

Notes:

1. Refer to *DFSORT/CMS User's Guide*, SC26-4361.
2. *dfsort* saves the GLOBAL TXTLIB setting, if any, and sets up a GLOBAL TXTLIB DFSRTLIB (unless the TXTLIB is already in the list of global TXTLIBS) before invoking DFSORT/CMS. The original TXTLIB setting is restored when DFSORT/CMS returns to *dfsort*. User programs must not interfere with this library.
3. Use *sort* to sort records that are longer than 32K. Be sure to have enough virtual storage to hold the entire file.
4. *syncsort* is a variant of *dfsort* that uses SYNCSORT TXTLIB instead of DFSRTLIB TXTLIB. It appears that SyncSort is not reentrant; running two *syncsort* stages concurrently is likely to have unpredictable results.
5. *vmsort* is a variant of *dfsort* that uses VMSLIB TXTLIB instead of DFSRTLIB TXTLIB.

diage4—Submit Diagnose E4 Requests

diage4 submits diagnose E4 requests to CP and writes the response to the primary output stream.

```

      CMS
      ►►—DIAGE4—◄◄
  
```

Type: Device driver.

Operation: Records from the primary input stream are used to build a parameter list for the diagnose E4 request. The CP response is written to the primary output stream. Refer to *z/VM CP Programming Services*, SC24-6272, for a complete description of input and output parameters.

Input Record Format: Input records contain a command verb that identifies the desired variety of diagnose E4.

```

      ►►—
      |   QLINK—word—devaddr—◄◄
      |   QMDISK—
      |
  
```

Issue a link query (subcode 0) or a minidisk query (subcode 1).

word The user ID to query.

devaddr The device name of the virtual device to be queried.

```

      ►►—FULLPACK—word—devaddr—devaddr—word—◄◄
  
```

Create a full pack minidisk overlay (subcode 2).

word The user ID who owns the minidisk.

devaddr The virtual device number of the minidisk in the specified user ID's directory entry.

devaddr The virtual device number of the created minidisk overlay.

word The link mode desired.

```

      ►►—FULLVOL—devaddr—
      |   number—
      |   devaddr—word—◄◄
  
```

devaddr The device number of the real device on which to place an overlay.

number Cylinder/block number to be verified for conflicts. The default is 0.

devaddr The virtual device number of the created minidisk overlay.

word The link mode desired.

digest

Output Record Format: The first four bytes contain the return code in binary. The next four bytes contain binary zeros. The remainder of the record contains the contents of the parameter list after the diagnose has been issued.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *diage4* does not delay the record.

Premature Termination: *diage4* terminates when it discovers that its output stream is not connected.

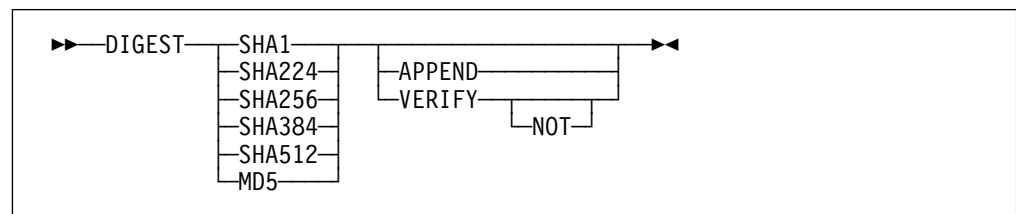
Notes:

1. FULLPACK and FULLVOL require CP directory OPTION DEVMAINT or equivalent ESM privilege. An unprivileged user may use QLINK and QMDISK only to enquire the characteristics of her own virtual machine or its directory entry.

Return Codes: *diage4* sets a nonzero return code only when it discovers syntactic errors in the input records. In particular, the return code does not reflect success or failure of the diagnose instructions issued.

digest—Compute a Message Digest

digest computes a message digest and optionally verifies the digest in the input record.



Type: Filter.

Syntax Description: One keyword is required; one is optional.

SHA1	Compute a 20-byte message digest according to RFC 3174.
SHA224	Compute a 24-byte SHA-2 message digest according to FIPS 180-3.
SHA256	Compute a 32-byte SHA-2 message digest according to FIPS 180-2.
SHA384	Compute a 48-byte SHA-2 message digest according to FIPS 180-2.
SHA512	Compute a 64-byte SHA-2 message digest according to FIPS 180-2.
MD5	Compute a 16-byte message digest according to RFC 1321.
APPEND	Secondary streams must not be defined. The digest is appended to the input record with appropriate padding.

:	VERIFY	Compute the digest of the beginning of the record and compare that with the digest present in the record as produced when APPEND is specified. Records that contain an invalid digest or are too short to include the complete digest are written in their entirety to the secondary output stream; for correctly verified records, the original record is written to the primary output stream; that is, padding and the digest are removed from the record.
:	NOT	The output streams are switched when NOT is specified; verified records are written to the secondary output stream.

Operation: The following applies when APPEND and VERIFY are omitted.

With only the primary streams defined, *digest* reads all its input and then produces a single digest on the primary output stream.

With secondary streams defined, the message on the primary input stream is passed to the primary output stream. At end-of-file on the primary input stream or whenever a record arrives on the secondary input stream, the current digest is written to the secondary output stream and the process is restarted. This mode is useful for batch signing of messages or files possibly for aggregation or transmission.

Streams Used: Secondary streams are optional when APPEND is omitted.

Record Delay: *digest* does not delay the record when APPEND is specified. When secondary streams are defined, it does not delay the record on the primary input stream being written to the secondary output stream or writing the record on the primary output stream relative to the record on the secondary input stream. Otherwise it delays the record until end-of-file.

Commit Level: *digest* starts on commit level -2. When APPEND is specified *digest* verifies that no secondary streams are defined. When VERIFY is specified, *digest* verifies that the secondary input stream is not connected. *digest* then commits to 0.

Premature Termination: *digest* terminates when it discovers that any of its output streams is not connected.

See Also: *cipher*.

Notes:

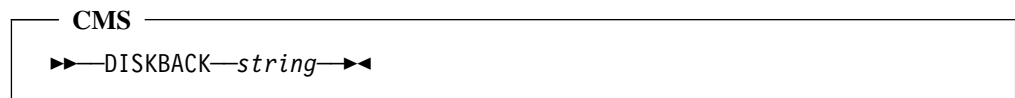
1. When the input data are in EBCDIC and you wish to send a signed ASCII message, you should translate appropriately and join all lines with carriage return and line feed before passing the record to *digest* APPEND.
2. *digest* discards records on the secondary input stream.
3. *digest* SHA1 and the four SHA-2 computations use hardware instructions when the message security assist feature is installed.
4. “Hardware instructions” should be taken to mean “Message-security Assist” and “Message-security Assist Extension 1” and “Message-security Assist Extension 2” facilities. *digest* specifically does not support Cryptographic coprocessors (“Integrated cryptographic facility”).
5. The SHA224 and SHA384 message digests are the truncated versions of SHA256 and SHA512 respectively.

- 6. The MD5 function is “derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm”.

diskback—Read a File Backwards

diskback is the generic name for a device driver that reads files into the pipeline, starting with the last record of the file and working forwards through the file.

Depending on the CMS level and the actual syntax of the parameters, *diskback* selects the appropriate device driver to perform the actual I/O to the file.



Type: Device driver.

Placement: *diskback* must be a first stage.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Minimum Release	Driver Used	Further Tests
1.2.0	<i>sfsback</i>	Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
(any)	<i>mdskback</i>	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.

Commit Level: Refer to the appropriate device driver.

Premature Termination: Refer to the appropriate device driver.

See Also: *diskslow* and *diskrandom*.

Examples: Refer to the appropriate device driver.

Notes:

1. *fileback* is a synonym for *diskback*.

diskfast—Read, Create, or Append to a File

diskfast is the generic name for a device driver that connects the pipeline to a file. When it is first in a pipeline, *diskfast* reads a file from disk; it treats a file that does not exist as one with no records (a null file). When it is not first in a pipeline, *diskfast* appends records to an existing file; a file is created if one does not exist.

Depending on the operating system and the actual syntax of the parameters, *diskfast* selects the appropriate device driver to perform the actual I/O to the file.

```
▶▶—DISKfast—string—◀◀
```

Type: Device driver.

Warning: *diskfast* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *diskfast* is unintentionally run other than as a first stage. To use *diskfast* to read data into the pipeline at a position that is not a first stage, specify *diskfast* as the argument of an *append* or *preface* control. For example, `|append diskfast ...|` appends the data produced by *diskfast* to the data on the primary input stream.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Operating System	Minimum Release	Driver Used	Further Tests
CMS	1.2.0	< <i>sfsfast</i>	The stage is first in the pipeline. Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
	1.2.0	>> <i>sfsfast</i>	The stage is not first in the pipeline. Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
	(any)	<i>mdskfast</i>	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.
MVS	(any)	< <i>mvs</i>	When first in a pipeline.
	(any)	>> <i>mvs</i>	When not first in a pipeline.

Record Delay: *diskfast* strictly does not delay the record.

Commit Level: Refer to the appropriate device driver.

Premature Termination: When it is first in a pipeline, *diskfast* terminates when it discovers that its output stream is not connected. Refer to the appropriate device driver.

See Also: `>`, `>>`, `<`, *diskback*, *diskrandom*, *diskslow*, *diskupdate*, *members*, and *pdsdirect*.

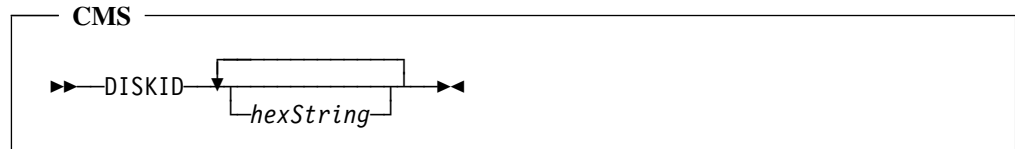
Examples: Refer to the appropriate device driver.

Notes:

1. *filefast* is a synonym for *diskfast*.
2. Use `<sfsfast` or `>>sfsfast` to access a file using a NAMEDEF that would be scanned by *diskfast* as a mode letter or a mode letter followed by a digit.

diskid—Map CMS Reserved Minidisk

diskid issues the DISKID function for a minidisk to obtain the offset to the first block of the reserved file.



Type: driver.

Syntax Description:

hexString Virtual device numbers for the minidisks to query.

Operation: *diskid* first processes the operand string, if present. It then reads its input and processes device numbers specified there.

Output Record Format: A sixteen byte record:

- Two bytes binary device number.
- Two bytes binary block size.
- Four bytes binary offset.
- Eight reserved bytes containing all zero bits.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *diskid* does not delay the record.

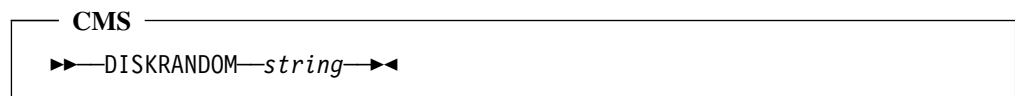
Premature Termination: *diskid* terminates when it discovers that its output stream is not connected.

See Also: *mapmdisk*.

diskrandom—Random Access a File

diskrandom is the generic name for a device driver that reads specified record ranges from a file into the pipeline.

Depending on the CMS level and the actual syntax of the parameters, *diskrandom* selects the appropriate device driver to perform the actual I/O to the file.



Type: Device driver.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Minimum Release	Driver Used	Further Tests
1.2.0	<i>sfsrandom</i>	Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
(any)	<i>mdskrandom</i>	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.

Commit Level: Refer to the appropriate device driver.

Premature Termination: Refer to the appropriate device driver.

See Also: *diskback* and *diskslow*.

Examples: Refer to the appropriate device driver.

Notes:

1. *filerandom* is a synonym for *diskrandom*.

diskslow—Read, Create, or Append to a File

diskslow is the generic name for a device driver that connects the pipeline to a file without buffering records internally. When it is first in a pipeline, *diskslow* reads a file from disk; it treats a file that does not exist as one with no records (a null file). When it is not first in a pipeline, *diskslow* appends records to an existing file; a file is created if one does not exist.

Depending on the operating system and the actual syntax of the parameters, *diskslow* selects the appropriate device driver to perform the actual I/O to the file.

▶▶—DISKSLOW—*string*—◀◀

Type: Device driver.

Warning: *diskslow* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *diskslow* is unintentionally run other than as a first stage. To use *diskslow* to read data into the pipeline at a position that is not a first stage, specify *diskslow* as the argument of an *append* or *preface* control. For example, `|append diskslow ...|` appends the data produced by *diskslow* to the data on the primary input stream.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Operating System	Minimum Release	Driver Used	Further Tests
CMS	1.2.0	<sfsslow	The stage is first in the pipeline. Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
	1.2.0	>>sfsslow	The stage is not first in the pipeline. Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
	(any)	mdskslow	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.
MVS	(any)	<mvs	When first in a pipeline.
	(any)	>>mvs	When not first in a pipeline.

Record Delay: *diskslow* strictly does not delay the record.

Commit Level: Refer to the appropriate device driver.

Premature Termination: When it is first in a pipeline, *diskslow* terminates when it discovers that its output stream is not connected. Refer to the appropriate device driver.

See Also: >, >>, <, *disk*, *diskback*, *diskrandom*, *diskupdate*, *members*, and *pdsdirect*.

Examples: Refer to the appropriate device driver.

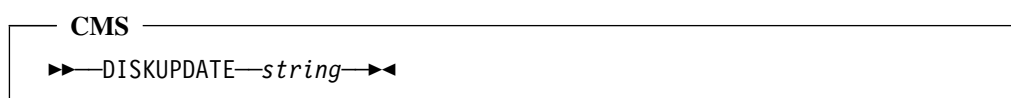
Notes:

1. *fileslow* is a synonym for *diskslow*.
2. Use <sfsslow or >>sfsslow to access a file using a NAMEDEF that would be scanned by *diskslow* as a mode letter or a mode letter followed by a digit.

diskupdate—Replace Records in a File

diskupdate is the generic name for a device driver that replaces records in a CMS file with data from the pipeline.

Depending on the CMS level and the actual syntax of the parameters, *diskupdate* selects the appropriate device driver to perform the actual I/O to the file.



Type: Device driver.

Placement: *diskupdate* must not be a first stage.

Syntax Description: An argument string is required.

Operation: The actual device driver to be used is selected based on the argument string:

Minimum Release	Driver Used	Further Tests
1.2.0	<i>sfsupdate</i>	Three or more words where the third word is not a mode letter or a mode letter followed by a digit.
(any)	<i>mfskupdate</i>	Two words or three words where the third is a mode letter or a mode letter followed by a digit. 7 through 9 are also considered mode numbers, even though they are rejected by CMS.

Record Delay: *diskupdate* strictly does not delay the record.

Commit Level: Refer to the appropriate device driver.

Premature Termination: Refer to the appropriate device driver.

See Also: *mfskupdate* and *sfsupdate*.

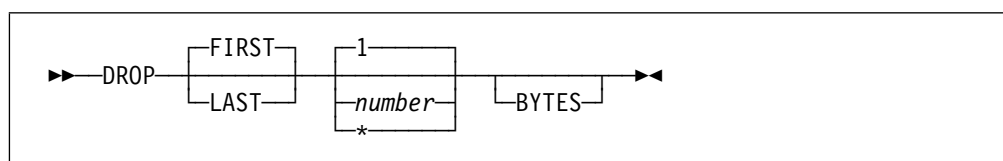
Examples: Refer to the appropriate device driver.

Notes:

1. *fileupdate* is a synonym for *diskupdate*.

drop—Discard Records from the Beginning or the End of the File

drop FIRST discards the first *n* records and selects the remainder. *drop* LAST selects records up to the last *n* and discards the last *n* records.



Type: Selection stage.

Syntax Description: The arguments are optional. Specify a keyword, a number, a keyword, or any combination.

FIRST Records are discarded from the beginning of the file. This is the default.

LAST Records are discarded from the end of the file.

number Specify the count of records or bytes to discard. The count may be zero, in which case nothing is discarded.

***** All records are discarded.

BYTES The count is bytes rather than records.

duplicate

Operation: When BYTES is omitted, *drop* FIRST copies the specified number of records to the secondary output stream, or discards them if the secondary output stream is not connected. It then passes the remaining input records to the primary output stream.

drop LAST stores the specified number of records in a buffer. For each subsequent input record (if any), *drop* LAST writes the record that has been longest in the buffer to the primary output stream and then stores the input record in the buffer. At end-of-file, *drop* LAST flushes the records from the buffer into the secondary output stream (or discards them if the secondary output stream is not connected).

When BYTES is specified, operation proceeds as described above, but rather than counting records, bytes are counted. Record boundaries are considered to be zero bytes wide. In general, the specified number of bytes will have been dropped in the middle of a record, which is then split after the last byte. When FIRST is specified the first part of the split record is discarded and the remainder is selected. When LAST is specified, the first part of the split record is selected and the second part is discarded.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. *drop* FIRST severs the secondary output stream before it shorts the primary input stream to the primary output stream. *drop* LAST severs the primary output stream before it flushes the buffer into the secondary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *drop* FIRST does not delay the record. When BYTES is not specified *drop* LAST delays the specified number of records. When BYTES is specified, *drop* LAST delays the number of records to needed for the specified number of bytes.

Commit Level: *drop* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *drop* terminates when it discovers that no output stream is connected.

Converse Operation: *take*.

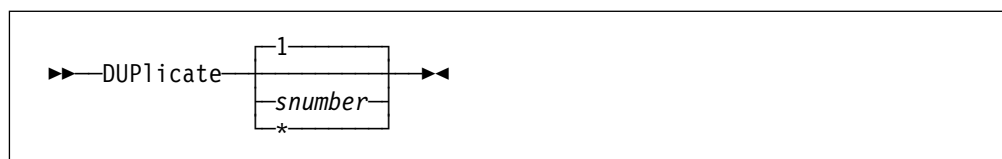
See Also: *flabel* and *tolabel*.

Examples: To remove the heading line from a command response:

```
pipe cp query rdr all | drop 1 | ...
```

duplicate—Copy Records

duplicate writes each input record into the pipeline one more time than the specified number.



Type: Filter.

Syntax Description: An operand is optional. Specify an asterisk or a number. The number must be -1 or larger. The number specifies the count of additional copies to write; the default is 1 (write each input record twice). An asterisk requests an infinite number of copies of the first input record.

Record Delay: An input record is consumed after all corresponding output records have been written.

Premature Termination: *duplicate* terminates when it discovers that its output stream is not connected.

See Also: *buffer*.

Examples:

```
!      pipe literal CMS Pipelines | split | dup | console
!      ▶CMS
!      ▶CMS
!      ▶Pipelines
!      ▶Pipelines
!      ▶Ready;
!      pipe literal CMS Pipelines | split | dup 2 | console
!      ▶CMS
!      ▶CMS
!      ▶CMS
!      ▶Pipelines
!      ▶Pipelines
!      ▶Pipelines
!      ▶Ready;
```

To create an infinite number of copies of the first input record:

```
... | dup * | ...
```

CMS Pipelines does not buffer the contents of the pipeline. Thus, though an infinite supply of records is at hand, records are not produced faster than they are consumed. That is, *duplicate* * provides an infinite supply of records, but it produces only one at a time.

duplicate * is often used to feed *delay* a stream of records to produce a record at a constant pace.

```
!      pipe literal +1 | dup * | delay | timestamp string /%T / | take 3 | ...
!                                     ... console
!      ▶14:50:31 +1
!      ▶14:50:32 +1
!      ▶14:50:33 +1
!      ▶Ready;
```

Notes:

1. *duplicate* -1 consumes all input and produces no output.
2. *duplicate* 0 copies records from the primary input stream to the primary output stream without any change.
3. To duplicate input records as groups, use *buffer* with a number to specify the number of copies.

elastic

elastic—Buffer Sufficient Records to Prevent Stall

elastic reads records from the input into a buffer and writes records from this buffer to the output in a way that does not prevent it from reading another record while it is writing a record. *elastic* may be used to avoid a stall in a multistream network.

When *elastic* has two input streams, the secondary input stream is assumed to be a feedback from the stages connected to the primary output stream.



Type: Gateway.

Operation: When the secondary input stream is not defined, *elastic* reads records as they arrive and writes them as they are consumed. It tries to minimise the number of records buffered inside.

When the secondary input stream is defined, *elastic* first passes the primary input stream to the primary output stream, buffering any records it receives on the secondary input stream. When the primary input stream is at end-of-file, *elastic* enters a listening mode on the secondary input stream. As long as it has records buffered, it writes to the primary output stream and reads what arrives at the secondary input stream and stores it in the buffer.

elastic flushes its buffer and terminates when the secondary input stream reaches end-of-file. *elastic* also terminates when the buffer is empty and there is no input record available after it has suspended itself to let all other ready stages run. At this point there should be no further records in the feedback loop; *elastic* terminates, because reading a further record would be likely to cause a stall.

Put another way: When *elastic* has a secondary input stream, it maintains a “to do” list. It adds items to do when records arrive on the secondary input stream and it deletes an item from the list when the corresponding output record is consumed. It terminates when the list is empty.

Streams Used: All records are passed from the primary input stream to the primary output stream before any records are passed from the secondary input stream to the primary output stream.

Record Delay: When the secondary input stream is defined, the records on the primary input stream are not delayed. *elastic* delays the records that it buffers; it may consume a record before writing it, even if the record can be written immediately.

Commit Level: *elastic* starts on commit level -2. It verifies that the secondary output stream is not connected, sets up a buffering stage, and then commits to level 0.

Premature Termination: *elastic* terminates when it discovers that its primary output stream is not connected.

See Also: *buffer* and *copy*.

Examples: To determine which files are embedded in a Script document:

```

/* See what files are imbedded (simplistic) */
'PIPE (name ELASTIC)',
  '|< document script',          /* Prime with main document */
  '| find .im_',                /* Look for these */
  '| e: elastic',              /* Maintain list of files to do */
  '| spec word 2 1 /script/ nextword', /* Generate file name */
  '| > document imbeds a',      /* Write result */
  '| getfiles',                 /* Read the contents of the files */
  '| find .im_',                /* Look for these */
  '| e:'                          /* Put on to-do list */

```

Notes:

1. Use *copy* when a delay of one record is sufficient. Use *buffer* when you know that the complete file must be buffered; for example, when another branch of the pipeline topology contains a *sort* stage.
2. It is expected that a case can be constructed that makes *elastic* with two input streams terminate before all data are processed.
3. It is unspecified how many records *elastic* buffers at a particular time. It may buffer more records than are required to avoid a stall.
4. *elastic* cannot cause a stall.

emsg—Issue Messages

emsg issues input lines as *CMS Pipelines* error messages under control of the message level setting and the standard VM message editing facility (the CP command SET EMSG). The message level controls how the message is delivered (the terminal, the console stack, or the output from *runpipe*) and what additional messages are issued to pinpoint the stage that issued the message. CP message editing can remove the message prefix or the message text, or it can suppress the message altogether.

►►—EMSG—◄◄

Type: Device driver.

Operation: Each line is issued with the MESSAGE pipeline command, which in turn passes the message on to CMS unless the pipeline set is under control of *runpipe* or the message level includes the bit for 256 which causes the message to be stacked instead. The line must have a 10- or 11-character prefix with module, message number (3 or 4 digits), and severity code.

Streams Used: Records are read from the primary input stream and written to the primary output stream. The input record is released after the message is issued.

Record Delay: *emsg* strictly does not delay the record.

Examples:

eofback

```
set msg text
▶Ready;
pipe literal dmsxxx123E Sample message| msg
▶Sample message.
▶... Issued from stage 2 of pipeline 1
▶... Running "msg"
▶Ready;
pipe literal dmsxxx123E Sample message| (nomsg 15) msg
▶Sample message.
▶Ready;
```

The second example shows how to disable the automatic messages; when one uses *msg* it is seldom interesting which stage issued the message.

Use the CP command SET MSG ON to display the message prefix:

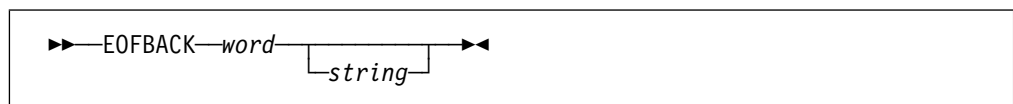
```
set msg on
▶Ready;
pipe literal dmsxxx1234E Sample message| (nomsg 15) msg
▶dmsxxx1234E Sample message.
▶Ready;
```

Notes:

1. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

eofback—Run an Output Device Driver and Propagate End-of-file Backwards

eofback passes the input records to the output. When the output record has been consumed, it is passed to the specified device driver, whose output stream is not connected.



Type: Control.

Syntax Description: Specify an output device driver and its arguments.

Record Delay: *eofback* strictly does not delay the record.

Premature Termination: *eofback* terminates when it discovers that its output stream is not connected.

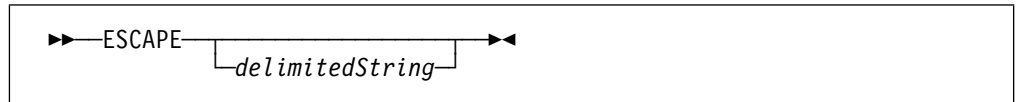
Examples: To write the file being processed to the terminal, without the *console* stage consuming all input:

```
... | eofback console | tolabel ...
```

Had *console* been used by itself, it would have written the entire input to the terminal. When used with *eofback*, it shows only those records that are consumed by the following stages, which typically will contain a partitioning selection stage.

escape—Insert Escape Characters in the Record

escape processes records to insert escape characters in front of characters that are specified as needing escape. This can be useful when building a pipeline where arbitrary argument strings are passed to stages.



Type: Filter.

Syntax Description: A delimited string is optional. The first character in the string is the escape character. Additional characters specify further characters to be escaped. A string consisting of double quotes, backward slash, and a vertical bar (`/\" | /`) is used by default.

Operation: In the input record, occurrences of characters in the argument string are prefixed with the first character of the argument string.

Record Delay: *escape* strictly does not delay the record.

Premature Termination: *escape* terminates when it discovers that its output stream is not connected.

See Also: *change*.

Examples:

```
pipe literal This: "a beautiful day" | escape | console
▶This: "\"a beautiful day\""
▶Ready;
```

Notes:

1. *escape* is a convenient substitute for a cascade of *change* filters.

fanin—Concatenate Streams

fanin passes all records on the primary input stream to the primary output stream, then all records on the secondary input stream to the primary output stream, and so on.



Type: Gateway.

Syntax Description: A blank-delimited list of stream identifiers and numbers is optional. Specify a list of numbers or stream identifiers, or both, to process input streams in a particular order. Only records from the specified streams are passed; other input streams are left intact. The default is to pass all records from the primary input stream to the primary output stream, then all records from the secondary input stream to the primary output stream, and so on.

Streams Used: Records are passed from all defined input streams or all specified ones; records are written to the primary output stream only.

Record Delay: *fanin* strictly does not delay the record.

Commit Level: *fanin* starts on commit level -2. It verifies that the primary output stream is the only connected output stream and then commits to level 0.

Premature Termination: *fanin* terminates when it discovers that its primary output stream is not connected.

Converse Operation: *fanout*.

See Also: *faninany*, *fanintwo*, and *gather*.

Examples: To write two files into the pipeline, one being upper cased and the other being lower cased:

```
/* CATTWO REXX */
parse arg fn1 ft1 fn2 ft2 .
'callpipe (end ?)',
  '|<' fn1 ft1,          /* Read first file          */
  '|xlate upper',      /* Uppercase it            */
  '|f:fanin',          /* Join inputs             */
  '|*:',               /* Pass on to next        */
  '|?<' fn2 ft2 ,     /* Read second file       */
  '|xlate lower',     /* Lowercase it           */
  '|f:'                /* Append to first        */
```

Notes:

1. *fanin* can cause a pipeline network to stall if two or more input streams originate in the same device driver; *faninany* cannot cause such a stall.
2. An *elastic* or a stage that buffers its file applied to all input streams except the primary will prevent a stall at the expense of storage.
3. Stream identifiers longer than four characters are truncated with a warning message.
4. Each input stream can be processed only once. *fanin* does not verify this. When the same stream is specified multiple times, subsequent references are ignored.
5. When mixing stream numbers and stream identifiers in *fanin*, be aware that streams identified with a stream identifier also have a stream number assigned by the scanner.

faninany—Copy Records from Whichever Input Stream Has One

faninany copies records from its input streams to the primary output stream. It reads records from whatever input stream has one ready. It is unspecified which stream is read next when two or more input streams have a record ready.



Type: Gateway.

Operation: When STRICT is specified, *faninany* ensures that it passes the record from the lowest-numbered stream that has a record available.

Streams Used: Records are read from all input streams; they are written to the primary output stream only.

Record Delay: *faninany* strictly does not delay the record.

Commit Level: *faninany* starts on commit level -2. It verifies that the primary output stream is the only connected output stream and then commits to level 0.

Premature Termination: *faninany* terminates when it discovers that its primary output stream is not connected.

See Also: *if*, *fanin*, *fanintwo*, and *gather*.

Examples: Assuming the input stream has examples between :xmp. and :exmp. tags at the start of a record, the following can be used to translate all examples to upper case. Because the stages in this pipeline do not delay the record, the order of the records is not affected by this pipeline. It does not matter whether they passed through the *xlate* stage or not.

```
/* UPXMP REXX */
'callpipe (end ?)',
  '|*:', /* Input stream */
  '|i:inside /:xmp./ /:exmp./', /* Take contents of examples */
  '|xlate upper', /* Do something on them */
  '|f:faninany', /* Re-merge the streams */
  '|*:', /* Pass them on */
  '|?i:', /* Short-circuit rest of file */
  '|f:'
```

Notes:

1. *faninany* cannot cause a stall.
2. Records from any one input stream appear in the output stream in the order in which they were read from that input stream, but they may be interspersed with records from other input streams. When multiple streams are being read, the relative order of the records from any two input streams is unspecified unless the input streams originate in a common selection stage or *fanout* (or similar) and the meshes in the pipeline topology consist entirely of stages that do not delay the record.
3. Depending on the number of input streams, STRICT may add significant overhead. Use it only when you can prove from the topology that it really is needed.

fanintwo—Pass Records to Primary Output Stream

fanintwo supports two input streams. It passes records from whichever input stream has one available to the primary output stream, giving priority to the secondary input stream.



Type: Arcane gateway.

Operation: When a record is available on *fanintwo*'s primary input stream, it suspends itself and then tests if there is a record on its secondary input stream as well. If there is, it passes the record from the secondary input stream and consumes it. *fanintwo* then goes back to look for another record on the secondary input stream to be passed before the record on the primary input stream, and so on.

fanintwo passes the record from the primary input stream to the output only when there is no record available on the secondary input stream. When *fanintwo* has written a record that originated on the primary input stream, it passes (and consumes) as many records from the secondary input stream as it can before it consumes the record on the primary input stream.

When AUTOSTOP is specified, *fanintwo* will pass any remaining records from the secondary input stream after end-of-file is detected on the primary input stream. This avoids a stall after processing all records on the primary input stream without the need for another stage like *gate* to terminate the feedback pipeline.

Streams Used: Records are passed from the primary input stream and the secondary input stream to the primary output stream. A record on the secondary input stream is passed in preference to one from the primary input stream.

Record Delay: *fanintwo* strictly does not delay the record it passes from the secondary input stream. It delays records it passes from the primary input stream by the number of records arriving on the secondary input stream before the record is consumed on the primary input stream.

Commit Level: *fanintwo* starts on commit level -2. It verifies that the primary output stream is the only connected output stream and then commits to level 0.

Premature Termination: *fanintwo* terminates when it discovers that its primary output stream is not connected.

See Also: *fanin*, *faninany*, and *gather*.

Examples: To generate an input to clear a TSO screen that is displaying the three asterisks:

```
'callpipe (end ? name FANINTWO.STAGE:56)',
  '?*.input.0:', /* Transactions */
  '|clr: fanintwo', /* Merge with automatic enters */
  '|dvmusi', /* Send to TSO */
  '|ldsfcfy', /* Figure out the 3270 data */
  '|f: strfind xle', /* MORE... */
  '"spec /DL0 ' /"', /* Generate ENTER automatically */
  '|elastic', /* No stall, please */
  '|clr:', /* Pass to TSO */
  '?f:', /* Data records */
  '|*.output.0:' /* To output */
```

The point of using *fanintwo* here is that an inbound record containing the attention identifier for the Enter key should be injected whenever TSO has written a line of three asterisks to indicate that more output is waiting. Were an input command issued instead, it would be ignored by TSO.

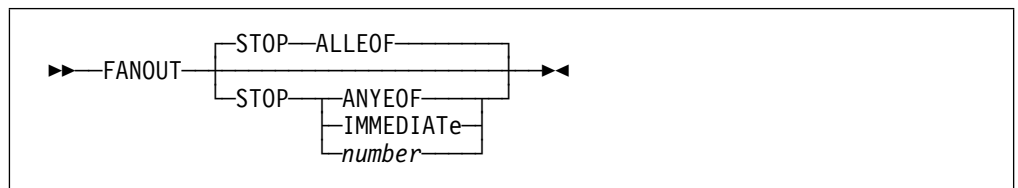
The workings of *dvmusi* and *ldsfcy* are “unspecified”; and they are not supplied with *CMS Pipelines*. *dvmusi* interfaces to the Logical Device Support Facility; *ldsfcy* classifies an inbound data stream to determine the state of the simulated terminal.

Notes:

1. *fanintwo* is useful to close an inner feedback loop where the feedback should have priority over the input from outside the loop.

fanout—Copy Records from the Primary Input Stream to All Output Streams

For each input record, *fanout* writes a copy to the primary output stream, the secondary output stream, and so on.



Type: Gateway.

Syntax Description: A keyword with its attendant option is optional to specify the conditions under which *fanout* should terminate. ALLEOF, the default, specifies that *fanout* should continue as long as at least one output stream is connected. ANYEOF specifies that *fanout* should stop as soon as it determines that an output stream is no longer connected. A number specifies the number of unconnected streams that will cause *fanout* to terminate. The number 1 is equivalent to ANYEOF. IMMEDIATE specifies that *fanout* stops as soon as it detects that an output stream is not connected and does not write the input record to any higher numbered output streams.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Records are written to all connected output streams.

Record Delay: *fanout* does not delay the record.

Commit Level: *fanout* starts on commit level -2. It verifies that the primary input stream is the only connected input stream and then commits to level 0.

Premature Termination: As determined by the argument. By default, *fanout* terminates when it discovers that no output stream is connected. *fanout* writes the input record to all connected output streams before it tests for the number of input streams at end-of-file. When IMMEDIATE is specified, *fanout* stops as soon as one output stream is not connected and does not write the record to any higher numbered output streams. *fanout* does not consume the record that causes it to terminate; this record has been written to all streams that are still connected.

See Also: *deal* and *fanoutwo*.

Examples: To generate a copy of the input record when the record is tested destructively:

```
'PIPE (end ? name FANOUT) ',
'|... ',
'|two: fanout',
'|p:  predselect',
'|... ',
'|?two:',
'|    xlate upper',
'|l:  locate /ANYCASE/',
'|p:',
'|... ',
'|?l:',
'|p:'
```

Notes:

1. *faninany* is normally used to gather records from a network of pipelines that is fed by *fanout*. Strictly, it is not the converse operation, because a cascade of *fanout* and *faninany* would generate as many copies of a particular record as there are streams between the two stages.

fanoutwo—Copy Records from the Primary Input Stream to Both Output Streams

fanoutwo is a specialised version of *fanout* designed to create a stream that can be passed to a device driver. Unlike a device driver, it propagates end-of-file backwards from the primary output stream to the primary input stream.



Type: Gateway.

Operation: *fanoutwo* passes the input record first to the primary output stream and then to the secondary output stream. It terminates when it receives end-of-file on the primary output stream; it shorts the primary input stream to the primary output stream when it receives end-of-file on the secondary output stream.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Two streams must be defined.

Record Delay: *fanoutwo* strictly does not delay the record.

Commit Level: *fanoutwo* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *fanoutwo* terminates when it receives end-of-file on the primary output stream.

Converse Operation: *faninany*.

See Also: *deal* and *fanout*.

Examples:

! The following pipeline shows how *fanoutwo* is used to probe the records flowing through
! the main pipeline, in this case to see just the first 3 records that match a certain selection.

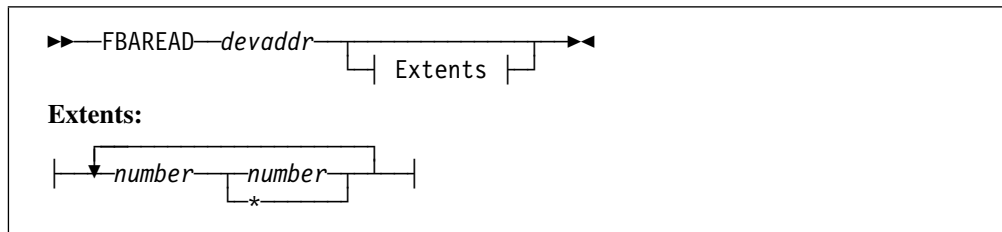
! The main pipeline continues when the secondary output stream of *fanoutwo* becomes
! unconnected.

```
! 'callpipe (end ?)',
!   '? *:',
!   '| o:fanoutwo',
!   '| rexx process',
!   '| *:',
!   '? o:',
!   '| pick w1 == ,*COPY,',
!   '| take 3',
!   '| console'
```

! While *fanout* with STOP ALLEOF continues to copy records to the primary output stream
! when the secondary output stream is disconnected, the difference with *fanoutwo* is that
! *fanout* is symmetrical and prevents end-of-file on the primary output stream to propagate
! back.

: ***fbaread*—Read Blocks from a Fixed Block Architecture Drive**

: *fbaread* reads blocks from an FBA disk and writes them to the primary output stream
: prefixed with origin information. The extents to be written can be specified as operands or
: in input records, or both.



: **Type:** Arcane device driver.

: **Syntax Description:**

: *devaddr* Specify the virtual device number of the disk to read. It must refer to an
: FBA device.

: Extents An extent is specified as the block number of the first block (decimal)
: followed by the number of blocks (decimal) or an asterisk which indi-
! cates to the end of the device. Multiple extents are specified by a
! multiple pairs of numbers. A single block is read when the count is
: omitted in the last extent. The first block on a device has number 0.

: **Output Record Format:** Each record contains a sixteen bytes prefix followed by one or
: more blocks of 512 bytes.

Pos	Len	Description
1	8	Check word: 'fplfba01'
9	4	Number of first block, binary.
13	4	Count of blocks, binary.
17		As many blocks of 512 bytes as specified by the block count.

fbawrite

: The blocks in an extent are written sequentially. Blocks from separate extents are never
: written to the same output record.

: **Record Delay:** *fbaread* does not delay the record.

: **Commit Level:** *fbaread* starts on commit level -10. It allocates an I/O buffer and then
: commits to level 0.

: **Premature Termination:** *fbaread* terminates when it discovers that its output stream is
: not connected.

: **Converse Operation:** *fbawrite*.

: **See Also:** *trackread*.

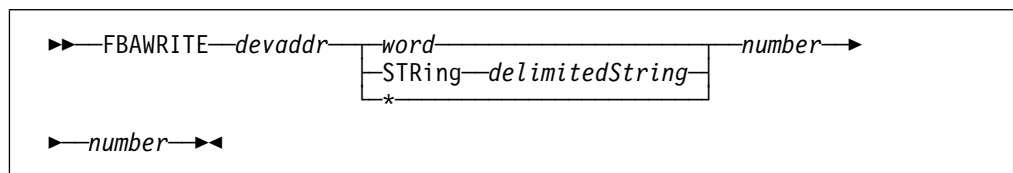
: **Examples:** To read the volume label of a disk:

```
: pipe fbaread 192 1 | substr 21.6 | console  
: ▶TMP192  
: ▶Ready;
```

: The label is at offset 4 in block number 1 on the device. The range above also includes
: the 16 byte header.

fbawrite—Write Blocks to a Fixed Block Architecture Drive

: *fbawrite* writes blocks to an FBA disk.



: **Type:** Arcane device driver.

: **Syntax Description:** Specify the device number, the current label on the device, and the
: first and last block in the writable extent.

: *devaddr* The virtual device number of the disk to write.

: *word* The current volume label on the device.

: STRING

: *

- A *word*, which is made upper case.
- The keyword STRING followed by a *delimitedString* for a label that contains characters in lower case or blanks.
- An asterisk to indicate that no label is present, for example, on a fresh temporary disk.

: *number* The first writable block number.

: *number* The last writable block number.

: The first and last blocks specify the extent into which blocks are written; the actual block
: address is obtained from the input record.

Operation: *fbawrite* verifies the device number and label as part of the syntax check.

Input Record Format: Each input record supplies a contiguous range of blocks to be written.

Each record contains a sixteen bytes prefix followed by one or more blocks of 512 bytes.

Pos	Len	Description
1	8	Check word: 'fplfba01'
9	4	Number of first block, binary.
13	4	Count of blocks, binary.
17		As many blocks of 512 bytes as specified by the block count.

The block range specified must be within the writable extent.

Streams Used: *fbawrite* passes the input to the output.

Record Delay: *fbawrite* does not delay the record.

Converse Operation: *fbaread*.

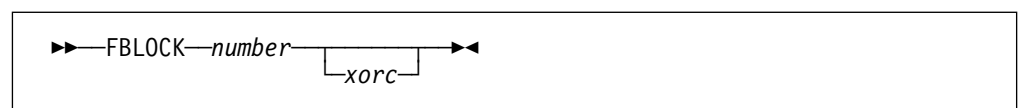
See Also: *trackwrite*.

Examples: To copy an FBA disk:

```
pipe fbaread 190 0 * | fbawrite 1190 * 0 72000
```

fblock—Block Data, Spanning Input Records

fblock writes records of fixed length that contain data from one or more input records. Data from an input record can span output records.



Type: Filter.

Syntax Description: A number is required; it specifies the length of output records. A second word is optional to specify the pad character for the last record. No padding is the default.

Operation: Conceptually, all records on the primary input stream are concatenated to a single logical record, which is then written as a number of records that have fixed length. An input record is, in general, spanned over output records (*block FIXED* does not allow this). The last output record is short when the length of the concatenated input is not an integral multiple of *number* and the pad character is omitted; the last record is padded to the record length with the pad character, if present.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

filedescriptor

Record Delay: *fblock* delays input records as required to build an output record. The delay is unspecified.

Premature Termination: *fblock* terminates when it discovers that its output stream is not connected.

See Also: *block* and *deblock*.

Examples: To count occurrences of characters in a file, first turn it into records of one character that *sort COUNT* can tally:

```
...| fblock 1 | sort count |...
```

To ensure that the length of the input records to *vchar* is a multiple of three:

```
...| fblock 3000 | vchar 12 16 |...
```

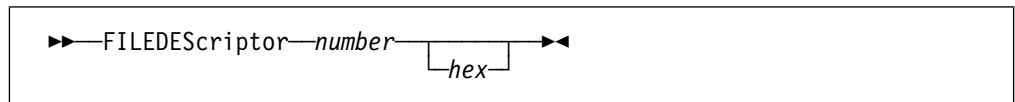
This ensures that characters are not spanned across record boundaries.

filedescriptor—Read or Write an OpenExtensions File that Is Already Open

filedescriptor connects the pipeline to a file that is managed by OpenExtensions.

When it is first in a pipeline, *filedescriptor* reads from the file until it receives zero bytes. When it is not first in a pipeline, *filedescriptor* appends the contents of its input records to the file.

The file must have been opened by the application before it issues the PIPE command; and the file descriptor must be closed by the application after the pipeline has completed. A program can use the callable services interface to open a file and to close a file descriptor.



Type: Device driver.

Syntax Description: A number is required; a hexadecimal storage address is optional.

The number specifies the file descriptor for the file to be read or written. File descriptors are integers from zero and up. Standard input is usually associated with file descriptor 0; standard output with 1; and standard error with 2. When a file descriptor is assigned, it gets the lowest unused number.

If it is present, the second operand specifies the storage address of an area into which *filedescriptor* stores additional information in the event of an error being reflected from OpenExtensions. The application should reserve thirty-two bytes for this area. Sixteen bytes are currently stored: The return code and reason code (each four bytes binary); and the name of the routine that returned the error (eight bytes, character). This operand should be specified only when the pipeline is issued from a program; results are unpredictable if this operand is specified from the command line or from a REXX program.

Streams Used: When *filedescriptor* is first in the pipeline, it writes records to the primary output stream. When *filedescriptor* is not first in a pipeline, it passes the input record to the output (if it is connected) after the record is written to the file.

Record Delay: *filedescriptor* strictly does not delay the record.

Commit Level: *filedescriptor* starts on commit level -2000000000. It verifies that the system does contain OpenExtensions and then commits to level 0.

Premature Termination: When it is first in a pipeline, *filedescriptor* terminates when it discovers that its output stream is not connected.

See Also: *hfs*, *hfsdirectory*, *hfsquery*, *hfsreplace*, *hfsstate*, and *hfsxecute*.

Notes:

1. When a return value of -1 is received from OpenExtensions and the second operand is omitted, *filedescriptor* issues error messages to identify the error before it terminates.
2. *stdin* is a convenience for *filedescriptor* 0.
3. *stdout* is a convenience for *filedescriptor* 1.
4. *stderr* is a convenience for *filedescriptor* 2.

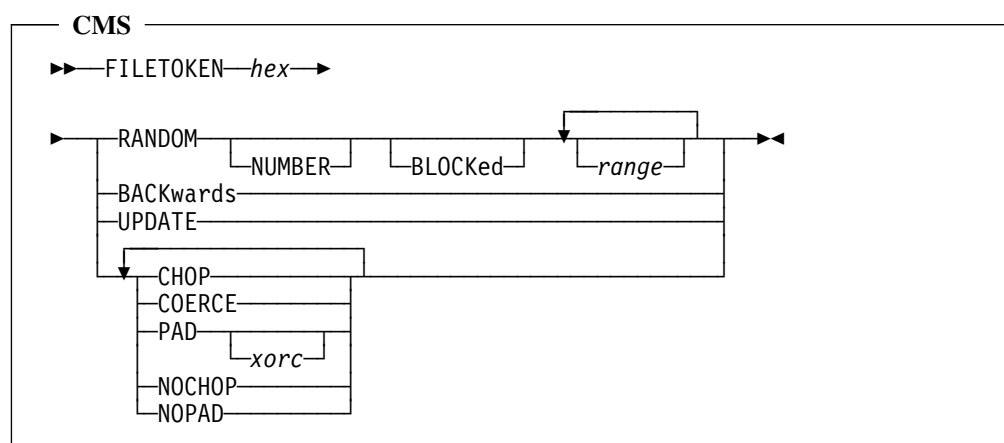
Return Codes: When the second operand is specified, the return code is the error number associated with the error. Otherwise the return code is the number of the error message issued.

filetoken—Read or Write an SFS File That is Already Open

filetoken connects the pipeline to a file that is managed by the Shared File System (SFS).

When it is first in a pipeline, *filetoken* reads from the file. When it is not first in a pipeline and RANDOM is omitted, it writes records to the file.

The file must have been opened by the application before it issues the PIPE command; and the file must be closed by the application after the pipeline has completed. You can use the CMS callable services to open the file.



Type: Arcane device driver.

Warning: *filetoken* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *filetoken* is unintentionally run other than as a first stage. To use *filetoken* to read data into the pipeline at a position that is not a first stage, specify *filetoken* as the argument of an *append* or *preface* control. For example, `|append filetoken ...|` appends the data produced by *filetoken* to the data on the primary input stream.

Syntax Description:

!	BACKWARDS	Read the file backwards. BACKWARDS is recognised only when <i>filetoken</i> is first in a pipeline.
!	BLOCKED	Write a range of records from the file as a single output record; the file must have fixed record format. BLOCKED is recognised only when <i>filetoken</i> is first in a pipeline.
	COERCE	A convenience for PAD CHOP. COERCE is recognised only when <i>filetoken</i> is not first in a pipeline.
!	CHOP	Truncate long input records to the logical record length of the file. The logical record length of a variable record format file is 65535 bytes. CHOP is recognised only when <i>filetoken</i> is not first in a pipeline.
!	NOCHOP	Do not truncate long records. Issue a message instead. NOCHOP is recognised only when <i>filetoken</i> is not first in a pipeline.
!	NOPAD	Do not pad short records. Issue a message on short records in fixed format files; ignore null records in variable record format files. NOPAD is recognised only when <i>filetoken</i> is not first in a pipeline.
!	NUMBER	Prefix the record number to the output record. The field is ten characters wide; it contains the number with leading zeros suppressed. NUMBER is valid only after RANDOM is specified.
!	PAD	Pad short records with the character specified. The blank is used as the pad character if the following word does not scan as an <i>xorc</i> . In a fixed format file, short records are padded on the right to the file's record length; in a variable record format file, a single pad character is written for a null record. PAD is recognised only when <i>filetoken</i> is not first in a pipeline.
	RANDOM	Read records randomly.
	UPDATE	Replace records randomly.

Input Record Format: When RANDOM is specified, input records contain a blank-delimited list where each word is a *range*.

When UPDATE is specified, the first 10 columns of an input record contain the number of the record to replace in the file (the first record has number 1). The number does not need to be aligned in the field. It is an error if an input record is shorter than 11 bytes.

The valid values for the record number depends on the record format of the file:

- Fixed For fixed record format files, any number can be specified for the record number (CMS creates a sparse file if required). An input record can contain any number of consecutive logical records as a block. The block has a single 10-byte prefix containing the record number of the first logical record in the block.
- Variable When the file has variable record format, the record number must be at most one larger than the number of records in the file at the time the record is written to it. The data part of input records must have the same length as the records they replace in the file.

Record Delay: *filetoken* strictly does not delay the record. When RANDOM is specified and *filetoken* is not a first stage, an input record that contains a single number is not delayed. Nor is an input record that contains a single range, when BLOCKED is specified.

Commit Level: *filetoken* starts on commit level -2000000000. It allocates a buffer and then commits to level 0.

Premature Termination: When it is first in a pipeline, *filetoken* terminates when it discovers that its output stream is not connected.

See Also: <, >, >>, *diskrandom*, and *diskupdate*.

Examples: To read records from a file for random update (error checking is omitted to make the example shorter):

```
/* Get private work unit */
call csl 'dmsgetwu sfsrc sfsreason workunit'
/* Open the file */
file='MY MASTER .INVENTORY'
intent='WRITE NOCACHE'
call csl 'dmsopen sfsrc sfsreason file' length(file),
        'intent' length(intent) 'filetoken workunit'
xtoken=c2x(filetoken)          /* Make printable          */
'PIPE',
  |'filetoken' xtoken 'random number' ranges,
  |'... ',
  |'filetoken' xtoken 'update'
pipeRC=RC
/* Return unit of work, which closes the file implicitly */
call csl 'dmsretwu sfsrc sfsreason workunit'
```

Notes:

1. Note that the file token is specified as an unpacked hexadecimal number. If you opened the file in a REXX program you must use the C2X built-in conversion function to make the file token printable.
2. You can use one *filetoken* stage to read records from a file and another one to replace or append to the same file, because the two stages use the same file token, as seen by SFS.

fillup—Pass Records To Output Streams

fillup passes records from the primary input stream to output streams as they “fill up”.



Type: Gateway.

Operation: Initially, records are passed to the primary output stream. When the primary output stream severed by its consumer (it propagates end-of-file backwards), *fillup* switches to the secondary output stream, and so on until all records are copied or there is only one stream left. In the latter case, *fillup* shorts the primary input to the last remaining output stream.

filterpack

Streams Used: Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *fillup* strictly does not delay the record.

Commit Level: *fillup* starts on commit level -2. It verifies that the primary input stream is the only connected input stream and then commits to level 0.

Premature Termination: *fillup* terminates when it discovers that no output stream is connected.

See Also: *deal* and *fanout*.

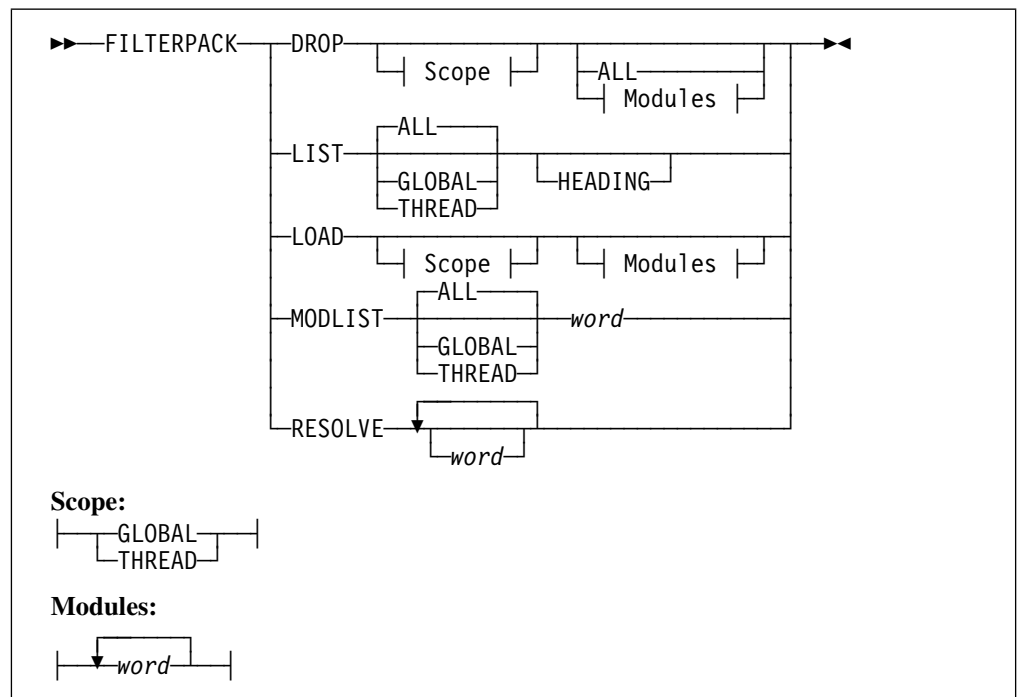
Examples:

The following pipeline shows where in the stream of records a filter stopped processing. When primary output stream of *fillup* is disconnected, the remaining input records are passed to the secondary output stream where the first record is shown that was not consumed by the process. is shown.

```
'callpipe (end \)',
  '\ *:',
  '| o: fillup',
  '| rexx process',
  '| *:',
  '\ o:',
  '| take',
  '| insert /Process stopped at /',
  '| cons'
```

***filterpack*—Manage Filter Packages**

filterpack loads and deletes filter packages and writes information about loaded filter packages.



Type: Arcane control.

Placement: *filterpack* LIST and *filterpack* MODLIST must be a first stage. For other variants, you may supply a list of modules or entry points on the primary input stream in addition to those specified as arguments.

Syntax Description:

DROP	Terminate use of filter packages. You can drop only filter packages that have been loaded by <i>filterpack</i> LOAD.
LIST	Write a line for each filter package currently loaded.
LOAD	Use filter packages. Specify the names of the modules to be loaded.
MODLIST	List contents of a filter package.
RESOLVE	Resolve an entry point and write the name of the containing filter package.
GLOBAL	Specify the global scope. This is the default for CMS and for the z/OS job step task. When used with <i>filterpack</i> LIST or <i>filterpack</i> MODLIST, the search for a filter package is restricted to the specified scope.
THREAD	Specify the thread local scope. This is the default for z/OS tasks other than the job step task. When used with <i>filterpack</i> LIST or <i>filterpack</i> MODLIST, the search for a filter package is restricted to the specified scope.

Output Record Format: For LIST, the record contains 7 words:

1. The name of the filter package.
2. The character G or L, which encodes the scope, possibly followed by a P for the PTF filter package.
3. The use count. The number of stages currently active in the filter package.
4. The address of the entry point table.
5. The address of the keyword look up table.
6. The address of the message text table.
7. The address of the user function table for *spec*.

filterpack MODLIST writes detailed information about the filter package specified including its entry points. The format of the detailed information is unspecified.

filterpack RESOLVE writes an output record for each word in the arguments. The output records consist of the name of the stage and, if the name is resolved, the name of the filter package that contains it. Both names are 8 byte fields, separated by a single blank. Built-in programs are considered to be in the filter package `builtin` (one leading blank), which is in lower case and has a leading blank.

Premature Termination: *filterpack* terminates when it discovers that its output stream is not connected.

Examples:

To see the list of currently loaded filter packages:

find

```
! pipe filterpack list | console
! ▶*PIPPTFF G      0 00EBF4F0 00000000 00000000 00000000
! ▶*PIPLOCF G      0 03F144F0 00000000 00000000 00000000
! ▶*PIPLOCF G      0 00000000 00000000 03F14610 00000000
! ▶*PIPSYSF G      0 00E9D530 00000000 00E9D820 00000000
! ▶*PIPIUOF G      0 03F0A548 00000000 03F0A6D8 00000000
! ▶FPLCPHR G       0 03DD6C30 00000000 00000000 00000000
! ▶SAMPFP G        0 03BD84F0 00000000 00000000 00000000
! ▶FPLPDF12 G      0 03C155E0 00000000 00000000 00000000
! ▶Ready;
```

To find whether the main pipeline or a filter package resolves a name:

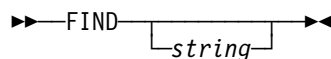
```
! pipe literal sort pattern foo | filterpack resolve | console
! ▶SORT builtin
! ▶PATTERN *PIPSYSF
! ▶FOO
! ▶Ready;
```

To list the contents of a filter package:

```
! pipe filterpack modlist sampfp | console
! ▶Module SAMPFP loaded dynamically
! ▶It contains no message table
! ▶It contains 1 entry points
! ▶Stage SAMPFILT at 03BD8510 flags 00000000.
! ▶It contains no function table
! ▶Ready;
```

find—Select Lines by XEDIT Find Logic

find selects records that begin with the specified string. It discards records that do not begin with the specified string. XEDIT rules for FIND apply.



Type: Selection stage.

Syntax Description: A string is optional. The string starts after exactly one blank character. Leading and trailing blanks are significant.

Operation: Input records are matched the same way XEDIT matches text in a FIND command (tabs 1, image off, case mixed respect):

- A null string matches any record.
- Blank characters in the string represent positions that must be present in the input record, but can have any value.
- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

find copies records that match to the primary output stream, or discards them if the primary output stream is not connected. It discards records that do not match or copies them to the secondary output stream if it is connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *find* strictly does not delay the record.

Commit Level: *find* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *find* terminates when it discovers that no output stream is connected.

Converse Operation: *nfind*.

See Also: *locate* and *nlocate*.

Examples: To select lines with 'a' in column 1 and 'c' in column 3:

```
pipe literal axc | literal abc | find a c | console
▶abc
▶axc
▶Ready;
pipe literal axc | literal abc| find a c | console
▶axc
▶Ready;
```

The first pipeline has two literal records that are both selected (the blank in the argument to *find* means “don’t care”). The argument string to *find* is four bytes in the second pipeline; thus, the record created by the second *literal* stage is not selected because it is only three bytes long.

To discard null records:

```
...|find |...
```

There are two blank characters after *find*. This means a record must have at least one character to be selected (but it does not matter what the character is). Because the two blank characters are easily missed, a popular alternative is to use *locate 1* (which selects the records that contain column 1).

To select records with a blank character, use an underscore character at that position in the pattern:

```
pipe literal a c | literal axc | literal a_c | find a c | console
▶a_c
▶axc
▶a c
▶Ready;
pipe literal a c | literal axc | literal a_c | find a_c | console
▶a c
▶Ready;
```

Notes:

1. All matching records are selected, not just the first one.
2. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the

comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

fitting—Source or Sink for Copipe Data

fitting is the space warp through which data are moved between a copipe program and the pipeline. In such an arrangement, the pipeline is set up by a separate program, which can be written in any language supported by CMS or z/OS. The program and the pipeline take turns in processing data.



Type: Gateway.

Syntax Description:

word Specify the fitting identifier.

Operation: When *fitting* is first in a pipeline, it accepts data from the copipe’s fitting request parameter list and injects these data into the pipeline.

When *fitting* is not first in a pipeline, it makes its input records available in the copipe’s fitting request parameter list. When the copipe has consumed the record, it is passed to the output stream (if it is connected).

Record Delay: *fitting* does not delay the record.

Notes:

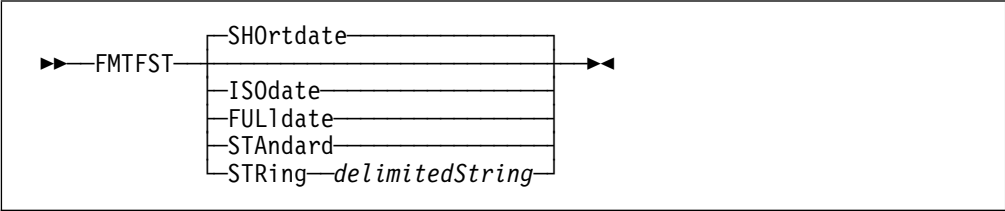
1. *fitting* can be used only when the pipeline is invoked with a FITG parameter token.

Publications:

PIPE Command Programming Interface, see “Additional Information, Download Site” on page xx.

fntfst—Format a File Status Table (FST) Entry

fntfst formats the contents of a File Status Table entry.



Type: Arcane filter.

Syntax Description: One keyword is optional to specify how the file’s timestamp should be formatted:

FULLDATE	The file's timestamp is formatted in the American format, with the century: 3/09/1946 23:59:59.
ISODATE	The file's timestamp is formatted with the century in one of the formats approved by the International Standardisation Organisation: 1946-03-09 23:59:59.
SHORTDATE	The file's timestamp is formatted in the American format, without the century: 3/09/46 23:59:59.
STANDARD	The file's timestamp is formatted as a single word in a form that can be used for comparisons: 19460309235959.
:	
STRING	Specify custom timestamp formatting, similar to the POSIX <code>strftime()</code> function. The delimited string specifies formatting as literal text and substitutions are indicated by a percentage symbol (%) followed by a character that defines the substitution. These substitution strings are recognised by <i>fmtfst</i> :
:	% A single %.
:	%Y Four digits year including century (0000-9999).
:	%y Two-digit year of century (00-99).
:	%m Two-digit month (01-12).
:	%n Two-digit month with initial zero changed to blank (1-12).
:	%d Two-digit day of month (01-31).
:	%e Two-digit day of month with initial zero changed to blank (1-31).
:	%H Hour, 24-hour clock (00-23).
:	%k Hour, 24-hour clock first leading zero blank (0-23).
:	%M Minute (00-59).
:	%S Second (00-60).
:	%F Equivalent to %Y-%m-%d (the ISO 8601 date format).
:	%T Short for %H:%M:%S.
:	%t Tens and hundredth of a second (00-99).

Input Record Format: The input record must be 64 bytes.

Output Record Format: Selected fields of the file status are formatted and written as a record: the file name, type, and mode; the record format and logical record length; the number of records and the number of disk blocks in the file; the date and time of last change to the file.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *fmtfst* strictly does not delay the record.

Premature Termination: *fmtfst* terminates when it discovers that its output stream is not connected.

Examples: To format a FST that has been stored as a file:

*f*rlabel

```
pipe cms listfile sample fst * ( format | console
▶FILENAME FILETYPE FM FORMAT LRECL
▶SAMPLE  FST      K1 V          64
▶Ready;
pipe < sample fst | fmtfst | console
▶FMTFST  STAGE   H1 V          70      80      1 12/05/92 16:36:5>
▶Ready;
pipe < sample fst | fmtfst iso | console
▶FMTFST  STAGE   H1 V          70      80      1 1992-12-05 16:36>
▶Ready;
pipe < sample fst | fmtfst full | console
▶FMTFST  STAGE   H1 V          70      80      1 12/05/1992 16:36>
▶Ready;
pipe < sample fst | fmtfst standard | console
▶FMTFST  STAGE   H1 V          70      80      1 19921205163657
▶Ready;
```

Notes:

1. *fmtfst* is designed to process the output from *affst* NOFORMAT, *state* NOFORMAT, and *statew* NOFORMAT.
2. SORTED is a synonym for STANDARD.

*f*rlabel—Select Records from the First One with Leading String

*f*rlabel discards input records up to the first one that begins with the specified string. That record and the records that follow are selected.



*f*rlabel copies records up to (but not including) the matching one to the secondary output stream, or discards them if the secondary output stream is not connected. It then passes the remaining input records to the primary output stream.

Type: Selection stage.

Syntax Description: A string is optional. The string starts after exactly one blank character. Leading and trailing blanks are significant.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. *f*rlabel severs the secondary output stream before it shorts the primary input stream to the primary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *f*rlabel strictly does not delay the record.

Commit Level: *f*rlabel starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *f*rlabel terminates when it discovers that no output stream is connected. Characters at the beginning of each input record are compared with the argument string. Any record matches a null argument string. A record that is shorter than the argument string does not match.

Converse Operation: *tolabel*.

See Also: *between*, *inside*, *notinside*, *outside*, and *whilelabel*.

Examples: To discard records on the primary input stream up to the first one beginning with the characters 'abc':

```
/* Skip to first record with label */
'callpipe *: | frlabel abc'
```

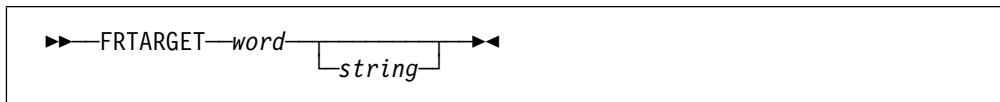
Because this invocation of *frlabel* has no secondary output stream, records before the first one beginning with the string are discarded. The CALLPIPE pipeline command ends when *frlabel* shorts the primary input stream to the unconnected primary output stream; the matching record stays in the pipeline.

Notes:

1. *fromlabel* is a synonym for *frlabel*.
2. Use *strfrlabel* with ANYCASE for caseless compare.
3. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

frtarget—Select Records from the First One Selected by Argument Stage

The argument to *frtarget* is a stage to run. *frtarget* passes records to this stage until the stage produces an output record on its primary output stream. The trigger record and the remaining input are then shorted to the primary output stream. Records that are rejected by the argument stage are passed to the secondary output stream (if it is defined).



Type: Control.

Syntax Description: The argument string is the specification of a selection stage. The stage must support a connected secondary output stream. If the secondary input stream to *frtarget* is connected, the argument stage must also support a connected secondary input stream.

Streams Used: Two streams may be defined.

Record Delay: *frtarget* does not add delay.

Commit Level: *frtarget* starts on commit level -2. It issues a subroutine pipeline that contains the argument stage. This subroutine must commit to level 0 in due course.

Premature Termination: *frtarget* terminates when it discovers that no output stream is connected.

Converse Operation: *totarget*.

See Also: *gate* and *predselect*.

Examples: To pass to the secondary output stream all records up to the first one that contains a string and to pass the remaining records to the primary output stream:

```
/* Frtarget example */
'callpipe (end ? name FRTARGET)',
  '|*:', /* Connect to input */
  '|f: frtarget locate /abc/', /* Look for it */
  '|*.output.0:', /* target and following */
  '|?f:',
  '|*.output.1:' /* Records before target */
exit RC
```

Notes:

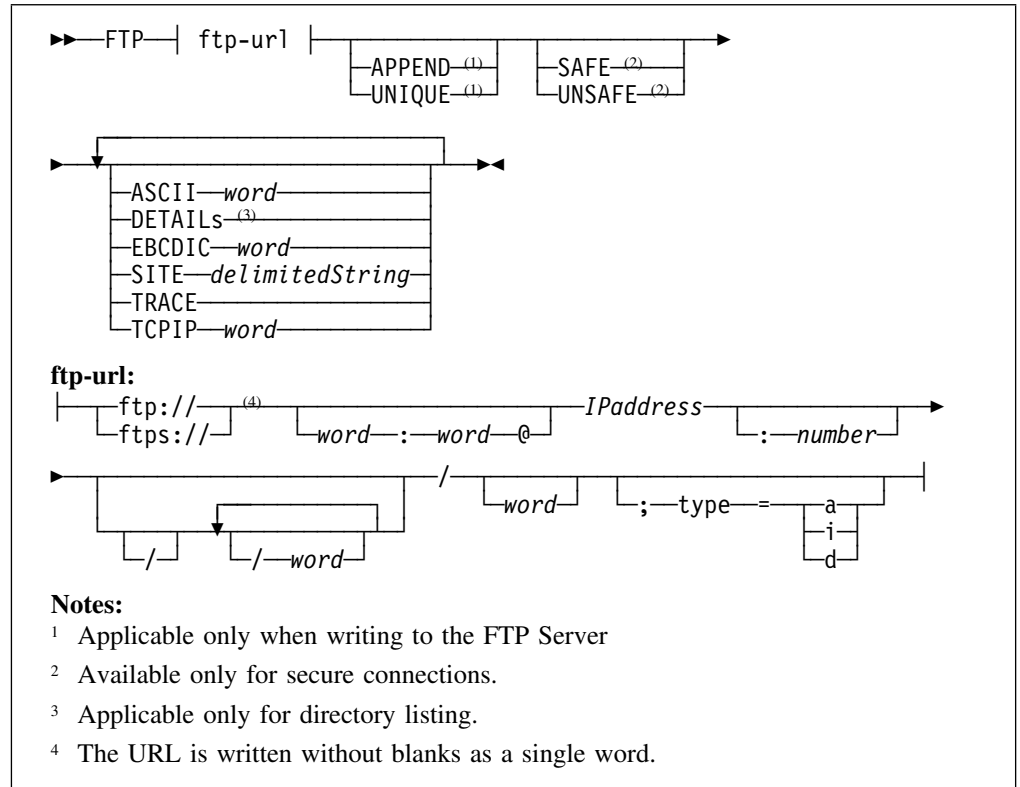
1. *fromtarget* is a synonym for *frtarget*.
2. It is assumed that the argument stage behaves like a selection stage: the stage should produce without delay exactly one output record for each input record; it should terminate without consuming the current record when it discovers that its output streams are no longer connected. However, for each input record the stage can produce as many records as it pleases on its secondary output stream; it can delete records. The stage should not write a record first to its secondary output stream and then to its primary output stream; this would cause the trigger record to be written to both output streams.

If the argument stage has delayed record(s) (presumably by storing them in an internal buffer) at the time it writes a record to its primary output stream, it will not be able to write these records to any output stream; the streams that are connected to the two output streams are severed when the argument stage writes a record to its primary output stream. End-of-file is reflected on this write. The records held internally in the argument stage will of necessity be lost when the stage terminates.
3. The argument string to *frtarget* is passed through the pipeline specification parser only once (when the scanner processes the *frtarget* stage), unlike the argument strings for *append* and *preface*.
4. *frtarget* is implemented using *fillup* and *fanoutwo*. The stage under test has only primary streams defined. The primary output stream is connected to a stage that reads a record without consuming it and then terminates. This means that any usage that depends on the secondary stream in the stage under test, will fail.

Return Codes: If *frtarget* finds no errors, the return code is the one received from the selection stage.

| *ftp*—Connect to an FTP Server and Exchange Data

| *ftp* connects the pipeline to an FTP Server using the File Transfer Protocol, optionally
| secured by z/VM System SSL. When it is first in the pipeline, *ftp* reads data from the FTP
| Server. When it is not first in the pipeline, *ftp* writes data to the FTP Server.



Type: Device driver.

Syntax Description: One operand is required to specify the address of the FTP Server and the protocol to use. The operand optionally specifies login credentials, the path and file name of the resource on the server, and the transfer type.

APPEND	Data written will be appended to an existing file on the FTP Server. The option is only available when the stage is not first in the pipeline. APPEND is mutually exclusive with UNIQUE.
ASCII	Specifies the codepage used by the host running the FTP Server. The default ASCII codepage is 850.
DETAILS	Request the system specific directory listing instead of the generic listing. The option is only valid when <i>ftp</i> is first in the pipeline and the qualifier <i>type=d</i> is specified.
EBCDIC	Specifies the codepage to use for the local system. The default EBCDIC codepage is 1047.
SAFE	Enables host name validation for the connection in <i>tcpclient</i> . SAFE is the default when the address of the FTP Server is specified as host name or host name with domain name. The option is only available for secure connections and is mutually exclusive with UNSAFE.
SITE	Specifies an optional SITE command to be issued. The SITE command is issued before data transfer, after navigating to the specified directory.
TRACE	Displays FTP commands and responses for diagnostic purposes.

TYPE=	Specifies the type of transfer as follows.
A	Transfer as text, translated from EBCDIC to ASCII with CRLF after each line.
I	Binary (<i>image</i>) transfer without translation.
D	Requests a directory listing rather than the contents of a file (only when <i>ftp</i> is first in the pipeline).
TCP/IP	Specifies the name of the TCP/IP stack (default TCP/IP).
UNIQUE	Data sent to the FTP Server will be written to a new file created on the server. The option is only available when writing to the FTP Server and is mutually exclusive with APPEND. Connect the secondary output stream of <i>ftp</i> to read a record that contains the unique name selected by the FTP Server.
UNSAFE	Disables host name validation for the connection in <i>tcpclient</i> . UNSAFE is the default when the address of the FTP Server is specified as dotted-decimal IP address. The option is only available for secure connections and is mutually exclusive with SAFE.

Operation: *ftp* uses a *tcpclient* stage to connect to the FTP Server. When *ftps://* is specified, a secure connection is established through z/VM System SSL according to the options defined in the z/VM System SSL configuration. The secure connection protects integrity and privacy of both the control connection and the data connection. When *ftp://* is specified, the connection is not secured with z/VM System SSL.

When the secondary input stream is connected, records are read from the secondary input stream to complement the login credentials in the URL. The records contain pairs of keyword and value.



USER	Specifies the user to authenticate the transaction.
PASS	The password used to authenticate the transaction.
ACCT	The account code specified is provided when requested by the FTP Server. The role of the account code depends on the FTP Server implementation and may be requested during logon or when accessing a file. On z/VM the account code is used as mini disk link password when no External Security Manager is used. Depending on whether <i>ftp</i> is reading or writing data, the READ or MULTI link password must be specified with ACCT.

Login is performed using the credentials provided in the URL or through the secondary input stream; *ftp* reports an error and terminates when the authentication fails or when the FTP Server configuration does not support the requested type of connection.

After login, *ftp* selects the specified directory on the FTP Server. When the specified directory does not exist, and *ftp* is writing a file, MKD commands are issued to create the

missing subdirectories, assuming subdirectories can be created, and enough levels of the specified path already exist to support this.

When SITE is specified, a SITE command is issued on the FTP Server. The SITE command is used by the FTP Server to provide services specific to the host system. For z/VM, a SITE command is often used to set the record format when creating a file. On z/OS, the SITE command can be used to pass allocation parameters when accessing the data set.

The position of *ftp* in the pipeline, and the optional TYPE qualifier in the URL determine the action performed by the FTP Server.

- When the URL specifies *type=d* a directory listing is requested from the FTP Server and written to the primary output stream.
- When *type=d* is not specified and *ftp* is first in the pipeline, the contents of the specified file is read from the FTP Server and written to the primary output stream.
- When *ftp* is not first in the pipeline, records are read from the primary input stream and written to the specified file on the FTP Server.

Streams Used: When the secondary input stream is connected, records are read to complement the login credentials. When the secondary output stream is connected, a record is written to it when *ftp* terminates after TCP/IP has reported an “ERRNO”.

Commit Level: *ftp* starts on commit level -10. It creates the pipeline with *tcpclient* to connect to the server, verifies that the TCP/IP connection is complete, and then commits to level 0.

Premature Termination: When it is first in a pipeline, *ftp* terminates when it discovers that its output stream is not connected.

ftp terminates when an error is reflected by the FTP Server or by TCP/IP. How it terminates depends on whether secondary output stream is connected or not.

When the secondary output stream is not connected, error messages are issued to describe the error and *ftp* terminates with a nonzero return code. When the secondary output stream is connected, a single record is written to the secondary output stream; *ftp* then returns with return code zero. For errors reflected by the FTP Server, the record contains the error messages reported by the FTP Server. When the error message from the FTP Server consists of multiple lines, these lines are concatenated with an ASCII linefeed that is typically translated to an EBCDIC X'25' character.

For errors detected by *tcpclient*, the record written contains the error number; the second word contains the symbolic name of the error number if the error number is recognised by *CMS Pipelines*. The assumption is that a REXX program will inspect the error number and decide whether it should retry the operation, discard the current transaction and retry, or give up entirely.

When the data transfer completes successfully and the secondary output stream is connected, a single record is written to the secondary output stream with the intermediate message reported by the FTP Server.

ftp also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *tcpclient*.

Examples:

To read the file SYSTEM LANGUAGE from the CMS S disk and display the contents as a single file.

```
pipe ftp ftp://127.0.0.1/maint.190/system.language | console
▶AMENG
▶UCENG
▶KANJI
▶Ready;
```

List the RPIVAL * files on the CMS Y disk.

```
pipe ftp ftp://127.0.0.1/maint.19e/rpival.*;type=d detail | console
▶RPIVAL  MAP      F      100      1      1 2016-08-24 10:23:48>
▶RPIVAL  MODULE   V      888      3      1 2016-08-24 10:23:48>
▶Ready;
```

To create a RECFM F file on a z/VM system, use the SITE option.

```
.. | ftp ftps://user:pass@127.0.0.1/prog.text site /fix 80/
```

Notes:

1. When the FTP transaction fails, the error message sent by the FTP Server is reported by *ftp*. To diagnose the problem, it may be helpful to repeat the transaction with the TRACE option to see all messages from the FTP Server. Note that the commands implemented by the FTP Server are not identical to the FTP commands used in the FTP client. When the TCP/IP connection fails with an ERRNO message, refer to *tcpclient* for additional information.
2. *ftp* supports explicit SSL/TLS in which the client initiates the SSL/TLS handshake with the AUTH TLS command (also referred to as STARTTLS protocol). *ftp* does not support an FTP Server using implicit SSL.
3. To avoid *CMS Pipelines* exposing login credentials, use the secondary input stream of *ftp* to pass that information to *ftp*. *ftp* does not use NETRC DATA to provide login credentials. When no login credentials are provided, *ftp* attempts an *anonymous* login.
4. When no TYPE qualifier is specified, and *ftp* is connecting to a FTP Server on z/VM or z/OS, data transfer is done in MODE B and TYPE E. This preserves the record boundaries in the data, and does not translate the data from EBCDIC to ASCII and back. Since MODE B transfer does not support records longer than 65535 byte, use TYPE=I and SITE to create a file with RECFM F and the required record length.
5. Since FTP is an ASCII service, login credentials, directories, and file name are translated from EBCDIC to ASCII (and back to EBCDIC when connecting to an FTP Server on z/VM or z/OS). Special characters should be *percent encoded* (% followed by the ASCII character value) to ensure correct operation. For example a space in the pass phrase should be encoded as %20 to represent the phrase as a single word.
6. When creating a file with UNIQUE, and the secondary output stream is connected, *ftp* writes a single record to the secondary output stream that contains the unique name assigned by the FTP Server. The contents of the record varies by FTP Server implementation. On z/VM, the unique name is based on the name specified in the URL. For example, this *ftp* stage was run twice to create a new file.

```
PIPE (end \) < some data
| f: ftpstage ftps://user:pass@host/abcdefgh.txt unique
\ f: | cons
```


On z/VM, the FTP Server first creates the file as requested (because it does not yet exist), and next with a file name that is based on the requested name.

```
125 Storing file 'ABCDEFGH.TXT' ( unique name )
125 Storing file 'ABCDEFG1.TXT' ( unique name )
```

A similar test on Linux shows the use of a file name suffix to create a new version of the file.

```
150 FILE: abcdefgh.txt
150 FILE: abcdefgh.txt.1
```

On a z/OS system, a similar pattern can be seen.

```
125 Storing data set RVDHEIJ.ABCDEFGH.TXT (unique name)
125 Storing data set RVDHEIJ.ABCDEFGH.TXT1 (unique name)
```

When writing an application that creates a unique file on some FTP Server, you may have to check the documentation or experiment to retrieve the name from the FTP Server response.

- Most FTP Server configurations provide a default “home” directory for the user after login. The path specified in the URL is relative to this initial directory. When the specified path starts with a “/” character, it is considered an absolute path; this shows in the URL as a double slash character after the IP address.
- When *ftp* is not first in the pipeline, and no input records are provided, *ftp* issues a DELE or RMD command to delete the specified file or directory. The example below deletes a file and a directory, respectively.

```
PIPE hole | ftp ftps://user:pass@host/testdata.txt
PIPE hole | ftp ftps://user:pass@host/vmsysu:./data/test/
```

To create an empty file on systems that support that, provide a null record as input to *ftp* and use the TYPE=I transfer mode.

Publications:

RFC 959 - File Transfer Protocol (FTP)

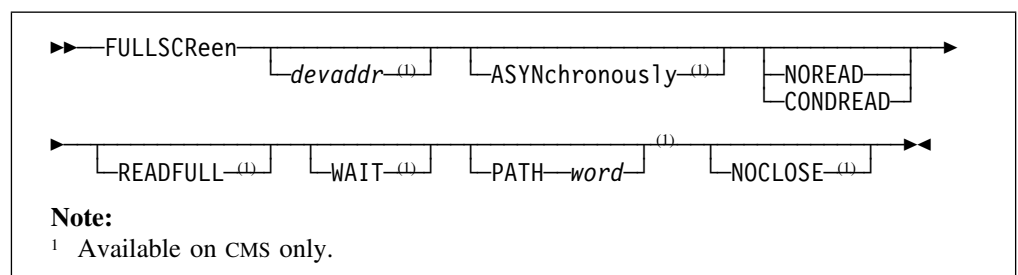
RFC 1579 - Firewall-Friendly FTP

RFC 2417 - Securing FTP with TLS

RFC 2428 - FTP Extensions for IPv6 and NATs

fullscr—Full screen 3270 Write and Read to the Console or Dialed/Attached Screen

fullscr writes its input records to a 3270 terminal and reads data from the terminal into the pipeline.



Type: Device driver.

fullscr

Placement: *fullscr* must not be a first stage.

Syntax Description: Options can be in any order. On z/OS, only the options NOREAD, CONDREAD, and READFULL are accepted.

A hexadecimal word specifies the virtual device address of a 3270 terminal attached or dialled to the virtual machine. The virtual machine console is used if no device address is specified. Do not specify the virtual device address of the console for the virtual machine.

The option NOREAD suppresses reads from the terminal for all input lines. CONDREAD suppress reads from the terminal when the control byte in column one of the input record includes the bit for X'01'.

READFULL specifies that the terminal is to be read with a read buffer operation; the default is to use a read modified operation.

A path may be specified for use by the CONSOLE macro with PATH; the default path is a unique name generated by *fullscr*. The path is closed at end-of-file unless NOCLOSE is specified.

The keyword ASYNCHRONOUSLY specifies that input records are written to the display as they arrive and that data are read from the terminal in response to an attention interrupt. This allows additional input records to be written to the display without waiting for the user to cause an attention interrupt. ASYNCHRONOUSLY is incompatible with NOCLOSE.

Operation: A write operation is performed after *fullscr* has read an input record unless you use CONDREAD and the input record is the single character X'02' or X'06'.

There are several ways to control the conditions under which *fullscr* waits for a response or reads the device:

- Use the option NOREAD to specify that you do not wish to wait for the terminal operator to enter a transaction. This is useful for an application that requires no operator intervention, for instance to update a status display or to write to a printer.
- Use CONDREAD to defer the decision to each individual data record. With this option, the device is read only if the rightmost bit (X'01') of the control byte is zero. With the CONDREAD option, the rightmost bit of the control byte set in the X'29' CCW is always zero.
- Specify ASYNCHRONOUS to read only in response to an attention interrupt from the terminal.
- Specify neither option to wait for an attention interrupt after each write, and then read the terminal.

A record containing the single byte X'00' is written to the output when CP signals that the terminal is in line mode at the time a full screen write is attempted (the write receives X'8E' status). Assuming that the screen is written without error, that ASYNCHRONOUSLY is omitted, and that *fullscr* does not wait for an attention interrupt, an output record containing the single byte X'02' is written as soon as the write completes.

A *solicited read* operation is performed without writing to the terminal if you specify CONDREAD and the input record contains a single X'02' or X'06'. The former causes a read buffer operation; the latter causes a read modified. The option READFULL is ignored for a solicited read.

When a solicited read is not performed and input is not suppressed, *fullscr* waits for an attention interrupt from the terminal, reads the inbound 3270 data stream, and writes it to the pipeline. Read modified is the default way of reading; read buffer is requested by the option READFULL. Write for positioning is not supported. The first byte of the output record is the attention ID (AID) character. The rest of the data depends on 3270 idiosyncrasies. Refer to the *3274 Description and Programmer's Guide*, GA23-0061, for details.

Input Record Format: The first position is a control byte that specifies how the record is to be processed. When the control byte includes the bit for X'20', the remainder of the input record consists of structured fields. When the control byte does not include the bit for X'20', additional data are an outbound 3270 data stream; the second byte of the record is the write control character (WCC); the remainder of the record is 3270 orders and data. Some of the bits in this control byte are defined by CP (the control field for CCWs with operation code X'29'); others are defined by *CMS Pipelines*:

100x xxxx	Erase. The screen is cleared and set to the default size (24 by 80). When no device address is specified (the virtual machine console is being used), CP sets full screen mode.
110x xxxx	Erase and write alternate. The screen is cleared and set to the alternate size. (The alternate size depends on the terminal; it is usually larger than 24 by 80.) Real 3277s do not support the alternate mode; the command will be rejected by CP. When no device address is specified (the virtual machine console is being used), CP sets full screen mode.
1x10 0000	Erase and write structured field. The screen is cleared. A write structured field operation is performed. When no device address is specified (the virtual machine console is being used), CP sets full screen mode.
0010 0000	Write structured field without erasing. A write structured field operation is performed. When no device address is specified (the virtual machine console is being used), the screen must be in full screen mode.
1x01 xxxx	Reflect the terminal break key to <i>CMS Pipelines</i> . This is available only when using the diagnose interface (DIAG58). The terminal break key function (normally Program Access key 1) is disabled. When the bit for X'10' is zero, the terminal break key can cause a CP break-in (that is CP takes over the terminal in line mode), unless break-in is set to guest control (CP TERMINAL BREAKIN GUESTCTL).
xxxx xxx1	Perform no read. When used with CONDREAD, this bit specifies that the write operation should be performed without waiting for an attention and without reading from the terminal.
0000 0010	Perform a read buffer. When no device address is specified (the virtual machine console is being used), the screen must be in full screen mode. CONDREAD must be specified to enable this; the input record must be one byte.
0000 0110	Perform a read modified. When no device address is specified (the virtual machine console is being used), the screen must be in full screen mode. CONDREAD must be specified to enable this; the input record must be one byte.
0000 000x	Perform a write without erase. When no device address is specified (the virtual machine console is being used), the screen must be in full screen mode.

A null input record is processed as X'0040', which indicates a write with a Write Control Character specifying no operation and no data.

Output Record Format: The output record consists of an inbound data stream from the terminal. The first character is an attention identifier (AID) or a pseudo-AID generated by *CMS Pipelines*. *fullscr* writes a one byte output record (a pseudo-AID) when it cannot or should not read from the terminal. If CP has put the terminal into line mode, *fullscr* does not attempt to recover. More than the present 3270 data stream may be needed to reformat the screen or set alternate screen size, or both.

The following values (in hexadecimal) are defined for the first byte of the output record:

- 00 CP has put the screen into line mode, as indicated by X'8E' unit status. The write was rejected; nothing was written to the screen.
- 01 CP has put the screen into line mode, as indicated by X'8E' unit status. Data were written to the screen, but the subsequent read failed because of a CP break-in.
- 02 Data have been written to the terminal; the read was suppressed as requested by NOREAD, or by CONDREAD with a control byte indicating that the read should be suppressed.
- 60 The screen has been read in response to a solicited read or a CP-generated attention due to a pending CP warning. Positions 2 and 3 contain the cursor position. For a solicited read, the data are in the format requested (read buffer or read modified). For a CP-generated attention, a read buffer has been performed if READFULL is specified; otherwise a read modified has been performed.
- 88 A structured reply has been read from the terminal. The first structured field begins in column 2.
- xx Data have been received from the terminal in response to an operator action. The AID identifies the key that generated the inbound transmission. Positions 2 and 3 contain the cursor position. When READFULL is in effect, the remaining data contain the complete screen buffer; when READFULL is not specified, the remaining data are modified fields.

Streams Used: Records are read from the primary input stream and written to the primary output stream. The input record is consumed after the screen is written and before the output record is written. This prevents a stall when both the input stream and the output stream from *fullscr* are connected to a REXX program that controls the panels shown on the display.

Record Delay: *fullscr* has the potential to delay one record.

Commit Level: *fullscr* starts on commit level -200000000. It ensures that the device is not already in use by another stage, allocates a buffer, and then commits to level 0.

Premature Termination: *fullscr* terminates when it discovers that its output stream is not connected. Use *hole* to consume output from *fullscr* that is not to be processed further.

See Also: *buildscr*, *fullscrq*, and *fullscrs*.

Examples: PIPDSCR EXEC shipped in PIPDSCR PACKAGE uses *fullscr*; see also RPQRY EXEC and 3270LOAD EXEC. SCRCTL REXX shipped in PIPGDSR PACKAGE shows how to manage a full screen display; the subroutine POPUP generates a panel.

To display a message on an unformatted screen (the left brace represents X'CO'):

```
pipe literal {BHit enter now! | fullscr
```

To poll the display to allow user input from a status display: send a record containing X'02' (read buffer) or X'06' (read modified) to *fullscr* to perform a solicited read. *fullscr* will then write an output record containing data read from the screen. The resulting attention ID is X'60' (a hyphen) if the user has not caused an attention interrupt. Note that such a read may show user input that is being entered, for which the user has not yet pressed an attention key.

```
/* Poll the user */
'PIPE strliteral x02 | fullscr condread path demo noclose | var response'
If left(response,1)=-'-' /* Action key pressed? */
  Then call user_input /* Input or CP break in */
```

Notes:

1. On a 3274 control unit supporting structured fields, you can issue any command as a 3270 data stream (3270DS) structured field. This lets you issue, for instance, an erase all unprotected command.
2. Improper data stream programming (lack of keyboard restore (X'02') in the Write Control Character) can get a 3270 terminal into a state where the keyboard is locked while *fullscr* is waiting for input from the terminal. Use the reset key on a locally attached terminal to enable keyboard entry. Use the ATTN key to gain access to CP from a terminal that is attached to an SNA control unit.
3. CP does not reflect errors in a 3270 data stream on terminals attached via PVM logical devices, VM/VCNA, or VM/VTAM. Thus, message 160 cannot be issued for such a terminal. VM/VCNA disconnects the terminal in this case; *CMS Pipelines* hangs until the terminal is reconnected. It may be necessary to enter CMS DEBUG and issue HX or to IPL CMS to recover.
4. *fullscr* supports an attached IBM 4224 printer using the Intelligent Printer Data Stream (IPDS). Request acknowledgement in the last structured field of a transmission so that an attention interrupt is always generated, be that for a NACK or as a solicited acknowledgement. This ensures that there is an inbound transmission after each write.
5. When CONSOLE is used, the macro interface requires that the first write to a path must erase the screen unless it is a write structured field. When the control byte of the first input record to *fullscr* indicates neither erase nor write structured field, and the control unit supports structured fields, *fullscr* generates a dummy write structured field order with a null 3270DS structured field to allow read or write without an initial erase.
6. When CONSOLE is used, the bit for X'10' in the control byte cannot be used to suppress the break-in function. Instead, issue the command TERMINAL BRKKEY NONE (or TERMINAL BRKKEY PF24 if the terminal has 12 program function keys) to suppress the break function.
7. *CMS Pipelines* performs I/O operations directly to the terminal when using the diagnose interface. Specify the WAIT option to make *fullscr* enter an enabled wait state when a unit check occurs. The screen is not reset after the error condition; you can enter test mode on the terminal and display the hardware control blocks with information about where your 3270 data stream is in error. Go back to the normal mode and log on to VM again. Press reset and clear the screen if you do not go into test mode. Enter the immediate command HW to get out of the enabled wait.
8. On z/OS, the only supported options are NOREAD and CONDREAD.
9. Though IBM 3270 terminals observe the protocols and orders specified in *IBM 3270 Information Display System, Data Stream Programmer's Reference*, GA23-0059, terminal

fullscrq

emulators and protocol converters in general have been observed to divert from this specification, in particular amongst the ones found in university environments. Though this may be regrettable, it is a fact of life, with which a pipeline programmer must cope.

fullscrq—Write 3270 Device Characteristics

fullscrq writes a line containing information about the terminal. This includes the device geometry and the reply to a structured query device, if the terminal supports write structured field queries.



Type: Device driver.

Placement: *fullscrq* must be a first stage.

Syntax Description: A device address is optional on CMS. The log on terminal is queried when no address is specified. No arguments are allowed on z/OS.

Operation: One record containing the terminal characteristics is written to the primary output stream.

Output Record Format: On CMS, the output record consists of sixteen bytes of information from diagnose 24 followed by the information returned by diagnose 8C. On z/OS, this information is synthesised from information provided by TSO, possibly augmented with the response to a device query.

Pos	Len	Description
1	2	Virtual device type. The log on terminal is stored as X'8000'; a dialled terminal shows the type of real device.
3	2	Virtual device status and flags. This field is X'0000' on z/OS.
5	1	Virtual device class. Always X'40' (graphics).
6	1	Virtual device type. X'04' for a 3277 display. X'01' for a 3278, 3279, or similar display. X'02' for a 3270-family printer. This field is always stored as X'01' on z/OS.
7	1	Device model number.
8	1	Line length.
9	1	“Real device terminal code for a local virtual console.” (This is the definition of the field; it is not clear what this really means.) This byte is stored as X'00' on z/OS.
11	2	Virtual device address on CMS; set to X'0000' on z/OS.
13	1	Condition code: X'F0' indicates condition code zero.
14	2	Reserved.

Pos	Len	Description
16	1	On z/OS, the rightmost byte of the attribute flags. Refer to the description of the GTTERM macro.
16	1	Terminal features flag byte: 1xxx xxxx Extended colour present. x1xx xxxx Extended highlighting present. xx1x xxxx Programmable symbol sets present. xxxx xx1x “3270 emulation.” For example, a terminal attached by an IBM 7171 protocol converter. This bit is not set on z/OS. xxxx xxx1 14-bit addressing allowed.
17	1	Number of partitions.
18	2	Number of columns.
20	2	Number of rows (lines).
22	v	The structured fields received in response to a Read Partition Query.

Commit Level: *fullscrq* starts on commit level -2000000000. It obtains the information required, in some cases by doing I/O to the terminal, and then commits to 0.

See Also: *fullscr* and *fullscrs*.

Examples: To determine the geometry of the console of the virtual machine:

```
! pipe fullscrq | spec 19.2 c2d 1 21.2 c2d next | cons
! ▶           80           24
```

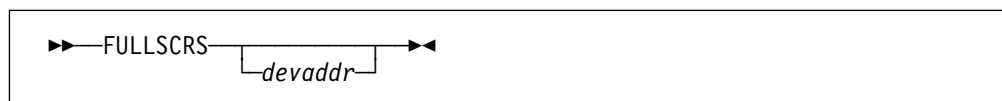
Notes:

1. CP caches the query reply; the contents of the output record reflect the status at the time the user logged on or last reconnected. Applications that need to know which symbol sets are currently loaded should perform their own query.

fullscrs—Format 3270 Device Characteristics

fullscrs processes the output record from *fullscrq* to extract screen geometry and other information useful to a REXX program that generates 3270 data streams for display on the specified terminal.

When *fullscrs* is not first in a pipeline it is assumed that the input record is in the format produced by *fullscrq*; when *fullscrs* is first in a pipeline it prefixes *fullscrq* to produce a record describing the terminal or the specified display (which must have been dialed in).



Type: Device driver.

Syntax Description: A device address is optional on CMS. The device (or the log on terminal) may be queried by *fullscrs* when there is insufficient information in the input record. The device address is verified only if *fullscrs* is first in a pipeline or if it needs to perform an I/O operation to the device.

fullscrs

Operation: If *fullscrs* is first in a pipeline, it prefixes a *fullscrq* to obtain the characteristics of the log in terminal.

Input Record Format: As defined for the output from *fullscrq*:

Pos	Len	Description
1	2	Virtual device type. The log on terminal is stored as X'8000'; a dialled terminal shows the type of real device.
3	2	Virtual device status and flags. This field is X'0000' on z/OS.
5	1	Virtual device class. Always X'40' (graphics).
6	1	Virtual device type. X'04' for a 3277 display. X'01' for a 3278, 3279, or similar display. X'02' for a 3270-family printer. This field is always stored as X'01' on z/OS.
7	1	Device model number.
8	1	Line length.
9	1	“Real device terminal code for a local virtual console.” (This is the definition of the field; it is not clear what this really means.) This byte is stored as X'00' on z/OS.
11	2	Virtual device address on CMS; set to X'0000' on z/OS.
13	1	Condition code: X'F0' indicates condition code zero.
14	2	Reserved.
16	1	On z/OS, the rightmost byte of the attribute flags. Refer to the description of the GTTERM macro.
16	1	Terminal features flag byte: 1xxx xxxx Extended colour present. x1xx xxxx Extended highlighting present. xx1x xxxx Programmable symbol sets present. xxxx xx1x “3270 emulation.” For example, a terminal attached by an IBM 7171 protocol converter. This bit is not set on z/OS. xxxx xxx1 14-bit addressing allowed.
17	1	Number of partitions.
18	2	Number of columns.
20	2	Number of rows (lines).
22	v	The structured fields received in response to a Read Partition Query.

Output Record Format: A record of blank-delimited words:

1. Number of lines.
2. Number of columns.
3. APL/TEXT flag:
 - 0 APL/TEXT not present.
 - 1 3278 APL/TEXT is present; use X'08' graphics escape orders.
 - 2 3277 APL/TEXT is present; use X'1D' escape sequences.

4. 1 if the terminal supports Erase/Write Alternate; 0 otherwise (typically a 3277).
5. 1 if Write Structured Field is supported.
6. 1 if Extended Highlighting is supported.
7. A two-character unpacked alias character for ROS (read only storage) font 1, or “no”. If present, this font contains the APL/TEXT character set.
8. The first halfword of the Coded Graphic Character Set Identifier for the ROS (read only storage) font 0 or a question mark if this information is not available.
9. The second halfword of the CGCSID, the code page, or a question mark if this information is not available.
10. The Coded Character Set ID for ROS (read only storage) font 0 or a question mark if this information is not available.
11. Encoded character information about the programmable symbol sets, if any. Semicolons separate the information about individual symbol sets. The fields for each symbol set are separated by hyphens:
 - a. The character set number.
 - b. The alias character associated with the character set, unpacked to two hex characters.
 - c. “r” for a ROS (read only storage) symbol set; “w” for a writable symbol set. The number of bit planes follows the r/w character.
 - d. If present, eight hexadecimal characters containing the unpacked CGCSID, consisting of a two-byte character set number and a two-byte code page number.
12. The contents of the colour reply, if one was received. Otherwise “no”.
13. The contents of the highlighting reply, if one was received. Otherwise “no”.
14. The contents of the reply mode reply, if one was received. Otherwise “no”.
15. The default and alternate cell size in the format `xx:yy/xx:yy`.

Record Delay: *fullscrs* strictly does not delay the record.

Commit Level: *fullscrs* starts on commit level -1. It verifies its arguments and then commits to 0.

See Also: *fullscr* and *fullscrq*.

Examples: To format the device information so that the character set information is on a line by itself:

```
!      pipe fullscrs | spec word 1.10 1 write word 11 1 write word 12-* 1 | cons
!      ▶24 80 1 1 1 1 F1 697 1047 ?
!      ▶0-00-r1-02B90417;1-F1-r1-03C30136
!      ▶41234567 0124 111
```

gate

gate—Pass Records Until Stopped

gate passes records from input streams other than the first to the corresponding output stream until a record arrives on the primary input stream, at which point it terminates. *gate* is used to terminate portions of a pipeline. For example, *gate* can be used to implement generalisations of the *flabel* and *tolabel* built-in programs, and to terminate device driver stages that do not terminate normally.



Type: Gateway.

Syntax Description: A keyword is optional.

Operation: *gate* issues the SELECT ANYINPUT pipeline command to wait for a record to arrive on any of its input streams. When a record arrives on the primary input stream, the primary input stream is shorted to the primary output stream *gate* then terminates.

When a record arrives on a stream other than the primary input stream and the option STRICT is specified, *gate* checks the primary input stream to see if a record is available before passing the record on other input streams to the corresponding output stream; *gate* then terminates without consuming the record.

When a record arrives on a stream other than the primary input stream and STRICT is omitted, *gate* may continue passing records while the primary input stream has a record ready; how many records depends on the pipeline dispatcher's strategy, which is unspecified. However, when *gate* is used to gate output from a selection stage, it may be known that there can be only one record available on all inputs at any one time; in this case you can avoid the overhead of the STRICT option.

If no record arrives on the primary input stream, *gate* terminates normally when all input streams are at end-of-file.

Streams Used: The primary input stream is shorted to the primary output stream. Records are passed from other input streams to the corresponding output stream.

gate ignores end-of-file on its primary output stream; it propagates end-of-file between the two sides of streams 1 and higher.

Record Delay: *gate* strictly does not delay the record.

Commit Level: *gate* starts on commit level -2. It allocates the resources it needs and then commits to level 0.

Premature Termination: *gate* terminates when it discovers that no output stream is connected.

See Also: *frtarget*, *predselect*, and *totarget*.

Examples: To terminate a *starmsg* and an *udp* stage when the immediate command STOP is issued:

```
'PIPE (end ? name GATE)',
  '|imcmd stop',          /* Wait for the command to stop */
  '|take 1',             /* Be sure to terminate IMMCMD */
  '|g: gate strict',    /* Terminate them */
  '?udp 69',
  '|g:',                /* through the gate */
  '|...',              /* Process UDP output */
  '?starmsg',
  '|g:',                /* through the gate */
  '|...',              /* Process STARMSG output */
```

A subroutine pipeline that terminates when it meets a record containing the string abc (in essence the function performed by *totarget*):

```
/* To target */
'callpipe (end ? name GATE)',
  '|*:',                /* Read input */
  '|l: locate /abc/',  /* Select the trigger */
  '|g: gate',         /* Force it to terminate */
  '?l:',              /* Records that don't contain string */
  '|g:',              /* Until one that does */
  '|*:',              /* Pass output */
exit RC
```

This subroutine pipeline passes records until a record is met that causes *locate* to write to its primary output. When this happens, *gate* terminates. This, in turn, severs both output streams from *locate*, and *locate* terminates without consuming the trigger record.

Notes:

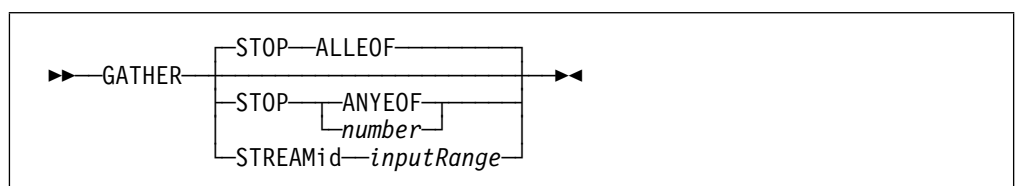
1. Use a cascade of *gate* stages instead of a single stage having more than two streams defined to avoid stalls when *gate* is blocked in writing a record. This also avoids the overhead of specifying STRICT.

gather—Copy Records From Input Streams

gather passes records from its input streams to the primary output stream. It can work in two ways: round robin or by stream identifier.

In the round robin mode, *gather* passes a record from the primary input stream to the primary output stream, then a record from the secondary input stream to the primary output stream, and so on. It returns to the primary input stream when it has passed a record from all defined streams.

In the stream identifier mode, *gather* reads a record from the primary input stream and extracts a stream identifier from a specified input range. If the stream identifier is null, blank, decimal zero, or an identifier that resolves to stream zero, the record is passed to the primary output stream. Otherwise a record is passed from the designated stream to the primary output stream.



gather

Type: Gateway.

Syntax Description: The keyword STOP or STREAMID is optional. The default is to continue until all input streams are at end-of-file.

STOP	ALLEOF, the default, specifies that <i>gather</i> should continue as long as at least one input stream is connected. ANYEOF specifies that <i>gather</i> should stop as soon as it determines that an input stream is no longer connected. A number specifies the number of unconnected streams that will cause <i>gather</i> to terminate. The number 1 is equivalent to ANYEOF.
STREAMID	Specify the input range that contains the stream identifier to be used to select the input stream to read from.

Operation: When STREAMID is omitted, streams at end-of-file are bypassed.

When STREAMID is specified and the record is passed from a stream that is not the primary input stream, the record on the primary input stream is consumed after the record that is passed.

Record Delay: *gather* strictly does not delay the record.

Commit Level: *gather* starts on commit level -2. It verifies that the primary output stream is the only connected output stream and then commits to level 0.

Premature Termination: *gather* terminates when it discovers that its primary output stream is not connected. *gather* STREAMID terminates as soon as it reaches end-of-file on any of its input streams.

Converse Operation: *deal*.

See Also: *fanin*, *faninany*, *fanintwo*, and *specs*.

Examples: *gather* is often used to gather the records that *deal* spread out to a set of parallel streams. Such a pipeline specification is usually built in a loop:

```
pipe='(end ?)'  
beg='*:*|D:deal'  
end='gather|*:'  
do i=1 to streams  
  pipe=pipe '|parallel process|G:' end  
  beg='?D:' /* Just the stream next time */  
  end='' /* Only first time */  
end  
'callpipe (end ?)' pipe
```

gather STREAMID is designed for the application that needs to process some records through a particular pipeline segment and others in some other way, where the processing involves elastics. Assume that records containing 1 in the first column must be sent to a server for processing; other records contain 0 in column one. To allow for some overlap, the server should run in parallel; there might even be several server threads (using *deal*):

```
'PIPE (end ? name GATHER.STAGE:94)',
'|?...|',
'|o: fanout',          /* Make a copy so we retain record on primary */
'|normal processing',
'|elastic',           /* buffer enough to wait for server */
'|g: gather streamid 1', /* Merge stuff */
'|...|',
'|?o:',
'|strfind /1/',      /* Take only those to server */
'|tcpclient 2555 9.55.5.13 sf onerresponse', /* Send to server */
'|g:'                /* Merge into stream */
```

When doing this type of pipeline networks, you should be careful not to flood the server. In this example this is ensured by the keyword `ONERESPONSE`, which makes `tcpclient` not delay the record; thus, `fanout` will be blocked if a second request arrives before the earlier one is processed.

Notes:

1. Input records must arrive in the order that `gather` reads them. Use `faninany` when the order cannot be predicted.

getfiles—Read Files

`getfiles` reads the contents of files into the pipeline. The files to read (as defined for `<`) are specified in input records. On CMS, this consists of the file name, the file type, and optionally the file mode. On z/OS, a single word or two words are acceptable.

►►—GETfiles—◄◄

Type: Device driver.

Operation: `getfiles` transforms each input record into a subroutine pipeline that is issued with `pipcmd` to read the file. A file is unpacked if it is packed.

Input Record Format: Input lines list files to be read into the pipeline. If present in the first seven columns, the string ' &1 &2 ' is ignored. (It is generated by the CMS command `LISTFILE` with the `EXEC` option.) After this string is removed, the first three words of the input line are passed as the arguments to a `<` stage, which reads the contents of the file into the pipeline.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: `getfiles` writes all output for an input record before consuming the input record.

See Also: `pipcmd`.

Examples: To read files whose names match a pattern and count the number of lines in them all:

```
pipe cms listfile ug* script h | getfiles | count lines | console
```

To count the aggregate number of words in the files specified in a `FILELIST`:

greg2sec

```
pipe < john filelist | getfiles | count words | console
```

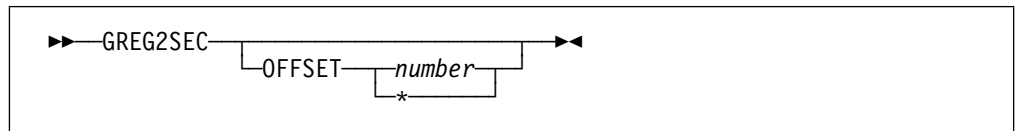
Notes:

1. *getfiles* terminates prematurely only if the pipeline stalls.

Return Codes: The return code from *getfiles* is the one from *pipcmd*, which in turn is the aggregate of the return codes from the CALLPIPE pipeline commands that issue the subroutine pipelines. A negative return code causes *pipcmd* to terminate; a positive return code indicates that all input records have been processed, though one or more are processed in error.

greg2sec—Convert a Gregorian Timestamp to Second Since Epoch

Convert a Gregorian timestamp to seconds since January first, 1970. A negative output number represents a time earlier than the epoch.



Type: Filter.

Syntax Description:

OFFSET A time zone offset is specified.

number Specify a number of seconds. The numerically largest acceptable value is 86399, as a time zone offset of 24 hours makes no sense. Positive values are east of Greenwich.

***** Use the time zone offset set for the host system.

Input Record Format: 14 characters without punctuation: `yyyymmddhhmss`.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *greg2sec* does not delay the record.

Premature Termination: *greg2sec* terminates when it discovers that its output stream is not connected.

Converse Operation: *sec2greg*.

See Also: *dateconvert*, *spec*, and *timestamp*.

Examples: The input record can be produced by *timestamp* with the proper format option:

```
pipe literal | timestamp string /%Y%m%d%H%M%S/ | greg2sec offset * | ...
... console
▶1588186233
▶Ready;
```

Notes:

1. The epoch started at 00:00:00 UTC on January first, 1970. This is the epoch used in UNIX systems.
2. LOCAL may also be specified to apply the local time zone offset.
3. A time zone offset of 86399 is not the same as one of -1.
4. For dates before year 1970, *greg2sec* ignores all issues as to whether the day actually occurred or the year existed at all.
5. The largest valid input timestamp is 99991231235959.
6. Leap seconds are not accounted for, as most UNIX systems also ignore this issue.

help—Display Help for CMS Pipelines or DB2

help processes requests for help. You can ask for help about:

- *CMS Pipelines* built-in programs, pipeline commands, syntax elements, and miscellaneous topics.
- *CMS Pipelines* messages. You can get help for a message issued by *CMS Pipelines* even if you do not know the message number; *CMS Pipelines* keeps track of the messages it issues.
- DB2 Server for VM “topics” and messages. *CMS Pipelines* also remembers the previous SQLCODES; you can get help for one without specifying its number.

Help about *CMS Pipelines* is stored either in the file PIPELINE HELPLIB or as standard CMS help files. Help about *TSO Pipelines* is stored in a partitioned data set which should be allocated to FPLHELP.

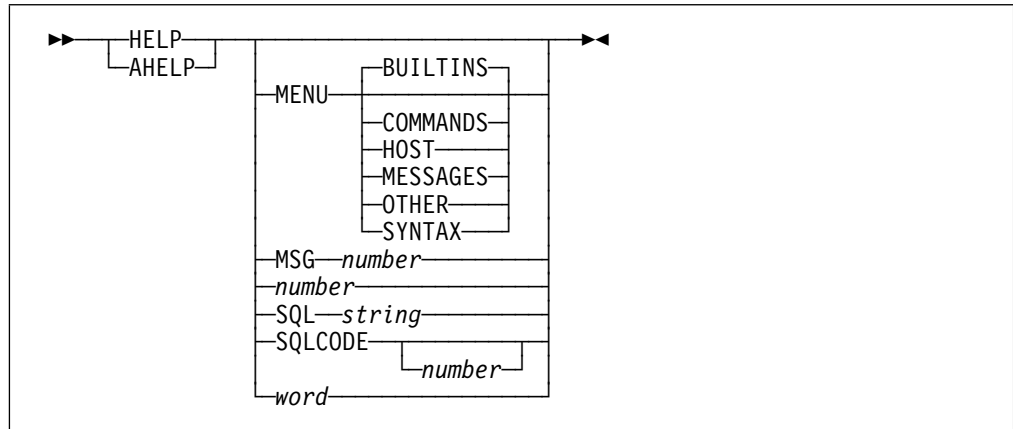
On CMS, help requests for *CMS Pipelines* topics are forwarded to standard CMS HELP in either of these cases:

- *help* is issued and the STYLE configuration variable is set to DMS.
- *help* or *ahelp* cannot find the file PIPELINE HELPLIB or the requested information is not in the library.

The remainder of this article applies to the case where PIPELINE HELPLIB is used or where a DB2 Server for VM topic is displayed.

When *help* finds the information to display, help is displayed on your terminal (using a normal XEDIT session on CMS) unless the primary output stream is connected; when the output stream is connected, the information is written to the pipeline rather than to the terminal.

help



Type: Service program.

Placement: *help* must be a first stage.

Syntax Description: The arguments are optional. When no arguments are specified, 0 is assumed; help is displayed for the last message issued.

MENU	Display a menu of <i>CMS Pipelines</i> topics. BUILTINS Show a menu of built-in programs. COMMANDS Show a menu of pipeline commands. HOST Show a menu of host commands related to <i>CMS Pipelines</i> . MESSAGES Show a menu of <i>CMS Pipelines</i> messages. OTHER Show a menu of miscellaneous topics, tutorials, <i>etc.</i> SYNTAX Show a menu of syntax variables.
MSG	Show help for the specified message. One or more blanks are optional between the keyword and the number.
<i>number</i>	(No keyword.) When the number is 10 or less, help is displayed for the last message issued (the number is 0), the second last message (the number is 1), and so on back through the memory of the last 11 messages issued. When the number is 11 or more, help is displayed for the message with that number. You must specify the MSG keyword to obtain help for messages 0 through 10.
SQL	Display DB2 Server for VM's help information about a particular topic. The topic may be a number (an SQLCODE) or the name of a SQL statement. This requires that you have connect privileges to DB2 Server for VM, that the help topics are loaded into the system tables, and that an access module has been generated by your installation; refer to help for <i>sql</i> .
SQLCODE	Display the help information for the last encountered nonzero SQLCODE (no number or 0) code without needing to remember what it was. "pipe help sqlcode 1" displays the help text for the second last return code received, and so on. <i>sql</i> remembers the last 11 nonzero return codes received from SQL.

word (The word is not a number.) Display help for the first member of the library (excluding messages) that the argument is an abbreviation of. This can be a built-in program, a pipeline command, a syntax variable, a host command, or a member that contains miscellaneous information. Note that you have to enter more than eight characters for some syntax variables (notably, *inputRanges* must be spelt out).

Operation: When MENU is specified, you can select other menus or members to be displayed with the cursor and press the Enter key or Program function key 1 (or 13). When CMS HELP is used, *help* calls standard CMS HELP for PIPE MENU; you are offered one menu for all built-in programs and all pipeline commands.

Tailoring help: The help library has character graphics and syntax diagrams using code points that display correctly on a 3270 with TEXT ON.

When help information is displayed in an XEDIT session (as opposed to using the system HELP), you can control how the XEDIT session is set up and you can change the help file as it is loaded into the XEDIT session.

When help information is displayed in an XEDIT session, the contents of the global variable PIPELINE_HELP_XEDIT_OPTIONS are automatically presented as options on the XEDIT command after a left parenthesis (which you should not add to the variable). This allows you to specify which profile to run or to suppress running a profile. A left parenthesis on the invocation of *help* sets the global variable permanently to the text following the parenthesis.

After the options are scanned for a left parenthesis, *help* looks for a REXX filter by the name *xithlp03*. If the file exists, it is called as a subroutine with the arguments *help* received and with its primary output stream connected to *help*'s primary output stream. The exit can inspect and set the default, if it so desires. A "good" default could be NOPROFILE NOMSG.

Two REXX filters are called, if they are present, to process lines before they are sent to XEDIT; the filters normally call a subroutine pipeline with an *xlate* filter. You can use other filters to change the help text to suit the character set in your terminal.

XITHLP02 REXX is used for lines in a menu; XITHLP01 REXX is called when sending help text to XEDIT. This example shows how to use only plus and hyphen for character graphics.

```
/* XITHLP01 REXX -- exit to translate lines for no TEXT feature */
'callpipe (name XITHLP01)',
  '|*:',
  '|xlate *-* ea-eb + ee-ef + ab-ac + bb-bc + bf - 8f + fa 4f',
  '|*:'
exit RC
```

Streams Used: If the primary output stream is connected, help is written to the primary output stream rather than being displayed. This applies even when CMS HELP is used.

Premature Termination: When the primary output stream is connected initially (and *help* writes the information to the output rather than displaying it), *help* terminates when the primary output stream becomes not connected.

Examples: Use *help* connected to *xedit* to display help information from SQL or from PIPELINE HELPLIB in the current XEDIT ring:

hfs

```
xedit new help s  
pipe help help | pad 80 | xedit  
set prefix off
```

To use the PIPEHELP XEDIT macro as the profile:

```
globalv setpl pipeline_help_xedit_options profile pipehelp
```

Note that you need to issue this command only once.

Notes:

1. When operation reverts to standard CMS HELP, severity codes must be specified with message numbers; built-in program names and pipeline commands must be spelt out to the minimum abbreviation.
2. Use *help* with a connected output stream to obtain help information when you wish to display it without additional XEDIT commands being issued.
3. As of *CMS Pipelines* level 1.1.10, the help library carries an index member, which relates the various names to the actual member names. As a result, more than eight characters can be used when requesting help for a topic.

Configuration Variables: For *help*, the configuration variable STYLE determines whether the PIPELINE HELPLIB is used or not. *help* goes directly to CMS HELP when the style is DMS. In the PIP and FPL styles *help* and *ahelp* are synonymous.

When *ahelp* is used in the DMS style and the file PIPELINE HELPLIB is not on an accessed mode, *ahelp* issues a dummy CMS HELP to make CMS access its help disk (where PIPELINE HELPLIB is stored) and then looks for the library one more time.

hfs—Read or Append File in the Hierarchical File System

hfs connects the pipeline to a byte stream file in the hierarchical file system. When it is first in the pipeline, it reads from the file; the file must exist. When *hfs* is not first in the pipeline, it appends to the file; the file is created if it does not exist.



Type: Device driver.

Syntax Description: The argument specifies the path to a file. When the first non-blank character is neither a single quote (') nor a double quote ("), the path is a blank-delimited word. Otherwise the path is enclosed in quotes in the REXX fashion; two adjacent quotes of the type that encloses the path represent a single occurrence of that quote. Only path names that contain blanks must be enclosed in quotes. A word that is not enclosed in quotes can contain quotes in the second and subsequent position; such quotes should not be “doubled up”.

Operation: When *hfs* is first in the pipeline, it reads bytes from the file into the pipeline. The number of bytes read at a time is unspecified.

When *hfs* is not first in the pipeline, it appends the contents of its input records to the file.

Streams Used: When *hfs* is first in the pipeline, it writes records to the primary output stream. When *hfs* is not first in a pipeline, it passes the input record to the output (if it is connected) after the record is written to the file.

Record Delay: *hfs* strictly does not delay the record.

Commit Level: *hfs* starts on commit level -2000000000. It verifies that the system does contain OpenExtensions, opens the file, and then commits to level 0.

Premature Termination: When it is first in a pipeline, *hfs* terminates when it discovers that its output stream is not connected.

See Also: *filedescriptor*, *hfsdirectory*, *hfsquery*, *hfsreplace*, *hfsstate*, and *hfsxecute*.

Examples: To read a file:

```
pipe hfs /u/john/.profile | deblock textfile | ...
```

To append to a file:

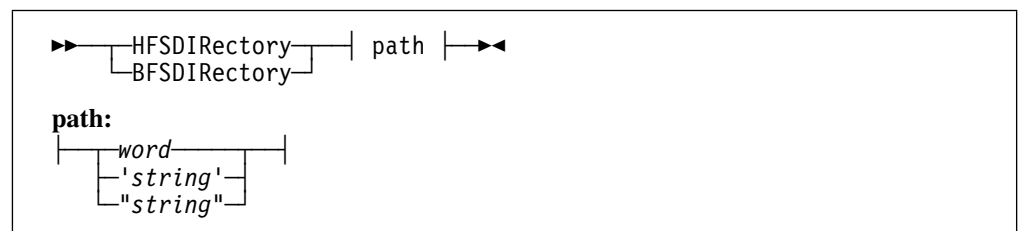
```
pipe literal startx | block 100 textfile | hfs /u/john/.profile
```

Notes:

1. Shell variables are not expanded; *hfs* does not run in the OpenExtensions environment.
2. When the first character of the path is not a forward slash (/), OpenExtensions prefixes the current working directory to the path.
3. OpenExtensions files are *byte stream files*. That is, they contain a number of bytes, but are not structured into records. Use *block* TEXTFILE to append newline characters to logical records that contain textual data.

hfsdirectory—Read Contents of a Directory in a Hierarchical File System

hfsdirectory reads the contents of a directory file and writes an output record for each entry in the directory. The directory must exist.



Type: Device driver.

Placement: *hfsdirectory* must be a first stage.

Syntax Description: The argument specifies the path to a directory, which must exist. When the first non-blank character is neither a single quote (') nor a double quote ("), the path is a blank-delimited word. Otherwise the path is enclosed in quotes in the REXX

fashion; two adjacent quotes of the type that encloses the path represent a single occurrence of that quote. Only path names that contain blanks must be enclosed in quotes. A word that is not enclosed in quotes can contain quotes in the second and subsequent position; such quotes should not be “doubled up”.

Commit Level: *hfsdirectory* starts on commit level 0. It verifies that the system does contain OpenExtensions, allocates a buffer, opens the file, and then commits to level 0.

See Also: *hfs*, *hfsquery*, and *hfsstate*.

Examples: To display the contents of the current working directory:

```
hfsdirectory . | console
▶.
▶..
▶data.file
▶READY
```

Notes:

1. Shell variables are not expanded; *hfsdirectory* does not run in the OpenExtensions environment.
2. When the first character of the path is not a forward slash (/), OpenExtensions prefixes the current working directory to the path.
3. OpenExtensions files are *byte stream files*. That is, they contain a number of bytes, but are not structured into records. Use *block* TEXTFILE to append newline characters to logical records that contain textual data.
4. Pass the output from *hfsdirectory* to *hfsstate* to obtain information about the file.

hfsquery—Write Information Obtained from OpenExtensions into the Pipeline

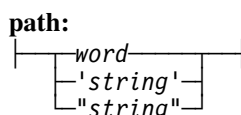
hfsquery obtains information from OpenExtensions. The information includes the current working directory, the contents of symbolic links, and information about the operating system.



Type: Device driver.

Input Record Format: A request code which may be followed by additional parameters. The request codes are:

- | | |
|-----------|---|
| pwd
cd | Query the path to the present working directory. This can be set by <i>hfsxecute</i> . |
| symlink | Query the contents of the symbolic link. The path to the symbolic link is specified as the second word of the line. It can be enclosed in quotes. |



uname Query the name of the Current Operating System. The output record contains information from the BPXYUSTN data structure.

Output Record Format: The output record for the pwd request contains the path to the current working directory.

The output record for the symlink request contains the contents of the symbolic link. That is, the name of the file being pointed to.

The output record for the uname request contains five tab-delimited fields (there are four tab characters in the record):

1. Name of implementation of operating system.
2. Name of this node within a communications network.
3. Release level.
4. Version level.
5. Name of the hardware type.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *hfsquery* strictly does not delay the record.

Commit Level: *hfsquery* starts on commit level -2000000000. It verifies that the system does contain OpenExtensions, allocates a buffer, and then commits to level 0.

Premature Termination: *hfsquery* terminates when it discovers that its output stream is not connected.

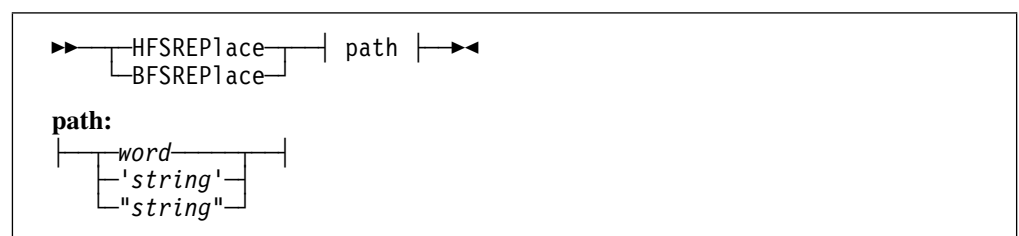
See Also: *hfsdirectory*, *hfsstate*, and *hfsxecute*.

Examples: To display the current working directory:

```
pipe literal pwd | hfsquery | console
```

hfsreplace—Replace the Contents of a File in the Hierarchical File System

hfsreplace replaces the contents of a file in a hierarchical file system with its input data. The file is created if it does not exist.



Type: Device driver.

Placement: *hfsreplace* must not be a first stage.

Syntax Description: The argument specifies the path to a file. When the first non-blank character is neither a single quote (') nor a double quote ("), the path is a blank-delimited word. Otherwise the path is enclosed in quotes in the REXX fashion; two adjacent quotes

of the type that encloses the path represent a single occurrence of that quote. Only path names that contain blanks must be enclosed in quotes. A word that is not enclosed in quotes can contain quotes in the second and subsequent position; such quotes should not be “doubled up”.

Operation: *hfsreplace* opens the file with the O_TRUNC flag, which causes the file to be truncated to a null file.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *hfsreplace* strictly does not delay the record.

Commit Level: *hfsreplace* starts on commit level -2000000000. It verifies that the system does contain OpenExtensions, opens the file, and then commits to level 0.

See Also: > and *hfs*.

Examples: To write a file:

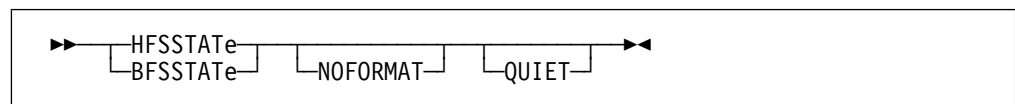
```
pipe literal Just one line | block 100 textfile | hfsreplace data.file
```

Notes:

1. Shell variables are not expanded; *hfsreplace* does not run in the OpenExtensions environment.
2. When the first character of the path is not a forward slash (/), OpenExtensions prefixes the current working directory to the path.
3. OpenExtensions files are *byte stream files*. That is, they contain a number of bytes, but are not structured into records. Use *block* TEXTFILE to append newline characters to logical records that contain textual data.
4. **Warning:** The file is opened on commit level -2000000000. Opening the file causes it to be truncated to a null file. This will destroy any existing data in the file, even if the pipeline is abandoned before reaching commit level 0.

hfsstate—Obtain Information about Files in the Hierarchical File System

hfsstate writes status information for files in a hierarchical file system. The output record contains information similar to that reported by the OpenExtensions shell’s ls command.



Type: Device driver.

Syntax Description: Two keywords are optional.

- | | |
|----------|---|
| NOFORMAT | Write the unformatted BPXSTAT data area to the output stream. |
| QUIET | Set return code zero, even when one or more files do not exist. |

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null input records are discarded. When a file is found, information about it is written to the primary output stream (if it is connected). When a file is not found, the input record is passed to the secondary output stream (if it is connected).

Record Delay: *hfsstate* strictly does not delay the record.

Commit Level: *hfsstate* starts on commit level -2000000000. It verifies that the system does contain OpenExtensions and that the secondary input stream is not connected and then commits to level 0.

Examples: To list the files in the present working directory:

```
pipe hfsdirect . | hfsstate | console
▶drwxrwxrwx  3 root    system      0 19940602143705 .<
▶drwxrwx--x  2 root    system      0 19940602143705 ..<
▶drwxrwxrwx  2 root    system      0 19940608234303 davidsen<
▶-rw-rw-rw-  1 root    system     12 19940609173256 tester<
▶READY
```

Notes:

1. Shell variables are not expanded; *hfsstate* does not run in the OpenExtensions environment.
2. When the first character of the path is not a forward slash (/), OpenExtensions prefixes the current working directory to the path.
3. OpenExtensions files are *byte stream files*. That is, they contain a number of bytes, but are not structured into records. Use *block* TEXTFILE to append newline characters to logical records that contain textual data.
4. The time of last modification is reported in a format that is suitable for sorting. The time is UTC; no time zone offset is applied.
5. On CMS, OpenExtensions directories do not contain the “dot” (.) and “dot-dot” (..) files; the example above was run on z/OS.

hfsxecute—Issue OpenExtensions Requests

hfsxecute reads requests from its input and passes them to OpenExtensions services. It passes the input record to the output, if it is connected.



Type: Device driver.

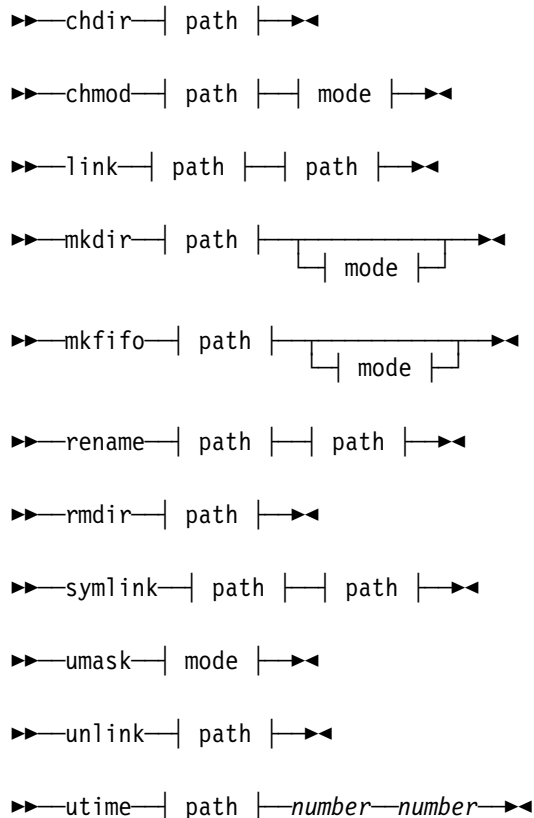
Placement: *hfsxecute* must not be a first stage.

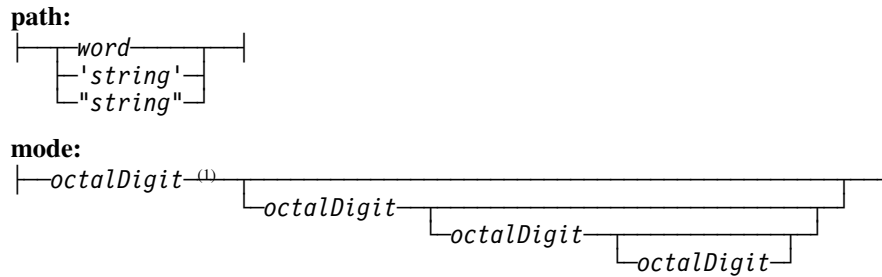
Operation: The requests are decoded as follows:

chdir	Change the current working directory to the one specified.
chmod	Change the mode of the file or directory.
link	Link the first path to the object specified by the second path. This is a “hard” link.

mkdir	Create a directory.
mkfifo	Create a named pipe.
rename	Replace the file or directory specified by the second path with the file or directory specified by the first path. If the target object does not exist, the existing file is renamed. Note the different meaning of “rename” in OpenExtensions.
rmdir	Erase a directory. The directory must be empty.
symlink	Create a symbolic link from the first path to the object named in the second path.
umask	Set the user file creation mask.
unlink	Delete a link to a file. When all links to a file have been deleted, the contents of the file are no longer accessible.
utime	Set the access and modification times for a file. The numbers are seconds since the epoch (midnight before January first, 1970); include leap seconds only when the operating system supports them.

Input Record Format: Each input record contains a request. Case is ignored in the first word; it is respected in path names.





Note:

¹ There are no blanks between the digits of a mode.

Streams Used: Records are read from the primary input stream and written to the primary output stream. *hfsxecute* passes the input record to the output if it is connected after it has passed the request to OpenExtensions.

Record Delay: *hfsxecute* strictly does not delay the record.

Commit Level: *hfsxecute* starts on commit level -2000000000. It verifies that the system does contain OpenExtensions and then commits to level 0.

Premature Termination: *hfsxecute* terminates as soon as a call to OpenExtensions sets the return value -1 (minus one), which indicates an error.

See Also: *hfsquery*.

Examples: To change the current working directory:

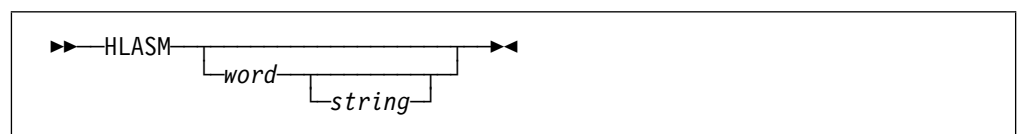
```
pipe literal chdir /u/john/oscar | hfsxecute
```

Notes:

1. Shell variables are not expanded; *hfsxecute* does not run in the OpenExtensions environment.
2. When the first character of the path is not a forward slash (/), OpenExtensions prefixes the current working directory to the path.
3. OpenExtensions files are *byte stream files*. That is, they contain a number of bytes, but are not structured into records. Use *block* TEXTFILE to append newline characters to logical records that contain textual data.

hiasm—Interface to High Level Assembler

hiasm invokes the High Level Assembler and then passes the primary input stream as the source program. The object module is passed to the primary output stream; the listing is passed to the secondary output stream; and the SYSADATA file is passed to the tertiary output stream.



Type: Filter.

Syntax Description:

<i>word</i>	Specify the file name of the input file. The default is “\$temp\$”.
<i>string</i>	Specify the parameter string for the High Level Assembler. This string must not contain exit specifications for any of the exits that are being used by <i>hiasm</i> .

Operation: *hiasm* builds a parameter string consisting of these items:

1. The first word of the arguments to *hiasm*, *word*, the file name.
2. A left parenthesis.
3. The words NODECK OBJECT. The word ADATA is added when the tertiary output stream is defined.
4. The exit specifications for the exits that are used. The exit for the object module is always declared; the exit for the listing and the SYSADATA file are declared only when the corresponding streams are defined.
5. The *string*. Thus, the argument string to *hiasm* can override the options specified previously.

hiasm then invokes the High Level Assembler as appropriate for the host system.

Whenever an exit is driven, *hiasm* supplies a record to the High Level Assembler or disposes of an output record. The High Level Assembler does not read from the input data set and it does not write to the output data sets for which an exit is declared.

Input Record Format: 80-byte card images.

Output Record Format: The primary output stream contains 80-byte card images. The secondary output stream contains the listing with whatever carriage control is specified and at whatever length the High Level Assembler supplies. The tertiary output stream contains the SYSADATA records; refer to the Programmer’s Guide or the data area (DSECT) for the format.

Streams Used: One to three streams may be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: The record delay is unspecified. In general, it is not possible to relate an output record to any particular input record.

Commit Level: *hiasm* starts on commit level -1. It builds the parameter list and loads the High Level Assembler into storage and then commits to level 0.

Premature Termination: Once the High Level Assembler has been called, it is not in general possible to make it terminate prematurely. *hiasm* must wait until the Assembler returns before it can terminate.

See Also: *hiasmerr*.

Examples: To assemble the current file in the XEDIT ring and bring the listing into the ring:

```

/* Assemble file */
'extract /fname'
':1'
'xedit' fname.1 'listing'
':1'
'delete *'
address command,
'PIPE (end ? name HLASM.STAGE:94)',
  '?xedit' fname.1 'assemble',
  '|pad 80',
  '|h: hlasm',
  '|hole',
  '?h:',
  '|xedit' fname.1 'listing'
':1'

```

This example ignores issues such as setting up macro libraries, setting the option to generate a listing file, and suppression of spurious XEDIT messages when deleting all lines of a file.

Notes:

- ! 1. High Level Assembler is a separate Licensed Product and is not part of z/VM. When
- ! High Level Assembler is not installed, *hlasm* will fail with an error message.
- 2. In addition to connecting the output streams, you must also enable the production of the corresponding file. The OBJECT option is specified by default; you can specify NOOBJECT to override it, but this will attract an assembler diagnostic message. LIST must be specified (or defaulted) to obtain any output on the secondary output stream.
- ! 3. At most one *hlasm* stage can be active at a time.
- ! 4. Be aware that when the secondary output stream is not defined, High Level Assembler
- ! will write the assembly listing to SYSPRINT. When no FILEDEF has been done, it will
- ! replace an existing LISTING file. To avoid that, define the secondary output stream.
- ! 5. The exits all use the same module name, FPLHLASX.

Publications: *High Level Assembler for MVS & VM & VSE: Programmer's Guide MVS & VM Edition*, SC26-4941. *High Level Assembler for MVS & VM & VSE: Language Reference MVS and VM*, SC26-4940. *High Level Assembler for MVS & VM & VSE: Installation and Customization Guide MVS & VM Edition*, SC26-3494.

Return Codes: Unless messages are issued, the return code is the one received from the High Level Assembler.

hlasmerr—Extract Assembler Error Messages from the SYSADATA File

hlasmerr processes the tertiary output stream from *hlasm* to relate error messages to input statements and macros.



Type: Filter.

Input Record Format: The SYSADATA file.

hole

Record Delay: *hlsasmerr* does not delay the record.

Premature Termination: *hlsasmerr* terminates when it discovers that its output stream is not connected.

See Also: *hlsasm*.

Examples: To write error messages to a file:

```
/* Do the assembly */
'PIPE (end ?)',
  '?... ', /* Construct input file */
  '|h: hlsasm', /* Assemble it */
  '|... ', /* Process object module */
  '?h:', /* Listing file here */
  '|... ', /* Process listing */
  '?h:', /* SYSADATA here */
  '|hlsasmerr', /* Make old-fashioned messages */
  '|>' fn 'errors a' /* Write to disk */
```

hole—Destroy Data

hole reads and discards records without writing any. It can be used to consume output from stages that would terminate prematurely if their output stream were not connected.

▶▶—HOLE—◀◀

Type: Device driver.

Streams Used: *hole* reads from all defined input streams; it does not write output. The output streams remain connected until *hole* reaches end-of-file on all its input streams.

Record Delay: *hole* delays all records until end-of-file.

Examples: To write two 3270 data streams that generate no response to the terminal:

```
/* HONK2 EXEC: Sound 3270 alarm twice */
address command
'PIPE (name HONK2)',
  '|literal' '00'x || 'D' ||,
  '|dup',
  '|fullscr noread',
  '|hole'
exit RC
```

fullscr writes a record containing X'02' to its output stream after it has written the input record to the terminal. Were *fullscr* to be at the end of the pipeline, it would terminate prematurely after the first write, because the output stream is not connected; *hole* ensures that the output from *fullscr* is consumed.

To issue the CMS command “zonq”, discarding terminal output and processing the lines stacked by the command after the command has ended:

```
pipe cms zonq | hole | append stack | ...
```

This works because the primary output stream from *hole* remains connected until end-of-file on its input; thus, *append* starts *stack* only after the ZONQ command has ended.

To discard records up to the next record that contains USER in the first four columns and a blank in column five:

```
/* Skip rest of the cards for the user */
'callpipe *: | tolabel USER | hole'
```

The *hole* stage consumes all output from *tolabel*. If it were omitted, the subroutine pipeline would terminate immediately without consuming any records.

Notes:

1. Connect *append* stages to multiple output streams from *hole* to start more than one device driver stage after a stream reaches end-of-file.

hostbyaddr—Resolve IP Address into Domain and Host Name

The input to *hostbyaddr* is a list of IP addresses to be resolved. For each word it determines the corresponding domain and host name; this information is written to the primary output stream. When the IP address cannot be resolved, the input word and additional information are written to the secondary output stream.

►►—HOSTBYADDR—◄◄

Type: Experimental resolver.

Operation: Processing in the event of an error during address resolution depends on whether the secondary output stream is defined or not. When there is only one stream, processing stops with an error message and a nonzero return code for severe errors, such as TCP/IP not being active; the inability to resolve the address is “reported” by not producing output. When the secondary output stream is defined, all errors are reported by writing a record on that stream.

Output Record Format: Records on the secondary output stream contain three or more words:

1. The input IP address.
2. The numeric ERRNO associated with the failure to resolve the address. (For example, “2053”.)
3. The symbolic ERRNO associated with the failure to resolve the address. (For example, “EUNKNOWNHOST”.)
4. The explanation of the ERRNO. (For example, “Unknown host”.)

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded.

Record Delay: *hostbyaddr* does not delay the record.

Commit Level: *hostbyaddr* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *hostbyaddr* terminates when it discovers that no output stream is connected.

hostbyname

Converse Operation: *hostbyname*.

See Also: *tcpclient*, *tcplisten*, and *udp*.

Examples:

```
! pipe (end ?) literal 129.33.199.226 129.33.199.255 | split | ...
! ... h: hostbyaddr | cons ? h: | insert ,Bad: , | console
! ▶www.vm.ibm.com
! ▶Bad: 129.33.199.255 2056 EIPADDRNOTFOUND IP address not found in ETC HOS>
! ▶Ready;
```

Notes:

1. On CMS, *hostbyaddr* uses RXSOCKET Version 2 or later for name resolution. As a consequence, the name is resolved using RXSOCKET rules. This implies that the file TCPIP DATA must be available and must point to the name server. RXSOCKET (unlike *CMS Pipelines*) uses the server virtual machine specified in TCPIP DATA.
2. RXSOCKET uses the file TCPIP DATA to determine the name of the TCP/IP service machine, the IP address of the name server, and so on.
3. RXSOCKET does not support hexadecimal components of a dotted-decimal number. It discards leading zeros in the components, and thus it treats an octal specification as a decimal one.

hostbyname—Resolve a Domain Name into an IP Address

The input to *hostbyname* is a list of hostnames (possibly qualified by their domain name). For each word it determines the corresponding IP address; this information is written to the primary output stream. When the domain or host name cannot be resolved, the input word and additional information are written to the secondary output stream.

▶▶—HOSTBYNAME—◀◀

Type: Experimental resolver.

Operation: Processing in the event of an error during name resolution depends on whether the secondary output stream is defined or not. When there is only one stream, processing stops with an error message and a nonzero return code for severe errors, such as TCP/IP not being active; the inability to resolve the name is “reported” by not producing output. When the secondary output stream is defined, all errors are reported by writing a record on that stream.

Output Record Format: Records on the secondary output stream contain three or more words:

1. The input domain name.
2. The numeric ERRNO associated with the failure to resolve the name. (For example, “2053”.)
3. The symbolic ERRNO associated with the failure to resolve the name. (For example, “EUNKNOWNHOST”.)
4. The explanation of the ERRNO. (For example, “Unknown host”.)

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded.

Commit Level: *hostbyname* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *hostbyname* terminates when it discovers that no output stream is connected.

Converse Operation: *hostbyaddr*.

See Also: *tcpclient*, *tcplisten*, and *udp*.

Examples:

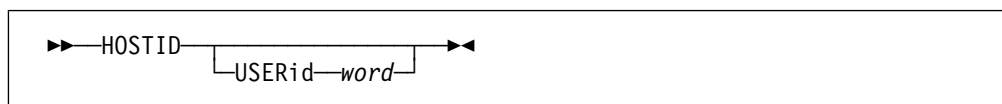
```
! To resolve host names and display the IP address:
!
! pipe literal gdlvm7 | hostbyname | console
! ▶9.56.214.105
! pipe literal www.vm.ibm.com | hostbyname | console
! ▶129.33.199.226
! ▶Ready;
! pipe literal www.mvs.ibm.com | hostbyname | cons
! ▶FPLTCR1142E Unable to resolve www.mvs.ibm.com (RXSOCKET error 2057 EHOST>
! ▶FPLMSG003I ... Issued from stage 2 of pipeline 1
! ▶FPLMSG001I ... Running "hostbyname"
! ▶Ready(01142);
!
! Use the secondary output stream to handle hostnames that could not be resolved:
!
! pipe (end ?) literal www.mvs.ibm.com | h: hostbyname | cons ? h: | ...
! ... insert ,Bad: , | console
! ▶Bad: www.mvs.ibm.com 2057 EHOSTNOTFOUND Host not found in ETC HOSTS >
! ▶Ready;
```

Notes:

1. On CMS, *hostbyname* uses RXSOCKET Version 2 or later for name resolution. As a consequence, the name is resolved using RXSOCKET rules. This implies that the file TCPIP DATA must be available and must point to the name server. RXSOCKET (unlike *CMS Pipelines*) uses the server virtual machine specified in TCPIP DATA.
2. RXSOCKET uses the file TCPIP DATA to determine the name of the TCP/IP service machine, the IP address of the name server, and so on.
3. There may be more than one IP address associated with a domain name. In this case, the output line contains blank-delimited IP addresses.

hostid—Write TCP/IP Default IP Address

hostid writes a single output record, which contains the default IP address of the TCP/IP system in dotted-decimal notation.



Type: Device driver.

hostname

Placement: *hostid* must be a first stage.

Syntax Description:

USERID Specify the user ID of the virtual machine or started task where TCP/IP runs. The default is TCPIP.

Commit Level: *hostid* starts on commit level -10. It connects to the TCP/IP address space and then commits to level 0.

Premature Termination: *hostid* terminates when it discovers that its output stream is not connected; *hostid* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

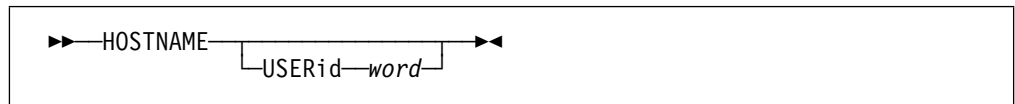
See Also: *hostname*.

Examples: To show the IP address of the default network interface:

```
pipe hostid | console
▶9.12.14.1
```

hostname—Write TCP/IP Host Name

hostname writes a single output record, which contains the host name of the TCP/IP system. The host name does not include the domain.



Type: Device driver.

Placement: *hostname* must be a first stage.

Syntax Description:

USERID Specify the user ID of the virtual machine or started task where TCP/IP runs. The default is TCPIP.

Commit Level: *hostname* starts on commit level -10. It connects to the TCP/IP address space and then commits to level 0.

Premature Termination: *hostname* terminates when it discovers that its output stream is not connected.

See Also: *hostid*.

Examples: To show the host name:

```
pipe hostname | console
▶WTSCPOK
```


httpsplit—Split HTTP Data Stream

httpsplit splits a hypertext transport protocol (HTTP) data stream into headers and contents. Headers are written to the primary output as individual records followed by a null line. The contents are written unchanged to the secondary output stream.



Type: Gateway.

Syntax Description: A keyword is optional.

EBCDIC Parse the header records in the EBCDIC domain. By default, header records are considered to be encoded in ASCII.

Operation: *httpsplit* processes its primary input stream in one of two modes, header or data. It starts in header mode.

In header mode, the input is considered a byte stream. A leading CRLF is discarded because an empty set of headers makes no sense. (This sequence is often sent erroneously by the client as a trailing CRLF on the previous data part.) Input is deblocked for carriage return and line feed until a null line is met. The deblocked records are written to the primary output stream.

The data part is processed in one of two ways. When the headers do not contain a line specifying content-length, the primary output stream is severed and all further input data are passed to the secondary output stream. Otherwise as many bytes as specified are passed to the secondary output. Processing then reverts to header mode.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream and written to the primary output stream.

Record Delay: *httpsplit* has the potential to delay one record on the primary output stream, because it can span input records. It strictly does not delay the record on the secondary output stream.

Commit Level: *httpsplit* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *httpsplit* terminates when it discovers that no output stream is connected.

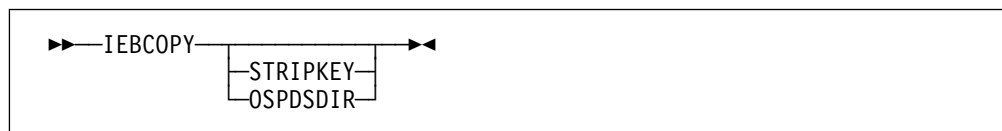
See Also: *urldeblock*.

Notes:

1. The data part is not joined or deblocked; raw input records are produced, possibly parts thereof for the first and last.
2. *httpsplit* does its processing for the primary output stream in the ASCII domain unless EBCDIC is specified.
3. In the ASCII domain CRLF is X'0d0a', whereas it is X'0d25' or X'0d15' in the EBCDIC domain.

iebcopy—Process IEBCOPY Data Format

iebcopy unravels data sets unloaded with the z/OS utility IEBCOPY. It is used by the sample OSPDS REXX to process a partitioned data set (PDS).



Type: Arcane filter.

Syntax Description: A keyword is optional. STRIPKEY specifies that the key portion of each block should be discarded. OSPDSDIR specifies that the key portion should be discarded and each block should be processed as a directory block of a partitioned data set.

Operation: *iebcopy* processes input records until it meets a disk block with key length zero and data length zero (an end-of-file block). An unloaded PDS contains several logical files: the first one is the directory of the PDS; the members follow in the order they are on disk (rather than the order of the directory, which is alphabetical).

Input Record Format: Input records contain one or more disk blocks. Each disk block consists of a 12-byte prefix followed by the key field (if any) and the data field. The prefix has the form FMBBCHHRKDD:

F A flag byte. The three leftmost bits must be zero.
M Ignored; should be zero (X'00').
BB Ignored; should be zero (X'0000').
CC Cylinder number.
HH Head number.
R Record number.
K Length of key part or zero.
DD Length of data part or zero for an end-of-file record.

Output Record Format: With no keyword specified, each output record contains the key and data parts of a disk block. With STRIPKEY, each output record contains the data part of a disk block. With OSPDSDIR, each output record contains a directory entry. The end-of-file record is not written to the output.

Record Delay: *iebcopy* does not delay the last record written for an input record.

Premature Termination: *iebcopy* terminates when it discovers that its output stream is not connected.

Examples: Refer to OSPDS REXX on the MAINT 193 disk.

Notes:

1. *iebcopy* returns when an end-of-file record is read. A control stage is needed to load all members of a PDS; refer to the sample file.
2. The first two logical records of the input data set must be discarded prior to *iebcopy*. The data set is usually written variable blocked spanned (RECFM=VBS); use *deblock* to obtain records in a format suitable for *iebcopy*:

```
pipe tape | deblock v | drop 2 | iebcopy ospdsdir | > pds drctry a
```

if—Process Records Conditionally

if runs a selection stage in an if/then or an if/then/else topology in conjunction with the stages up to its last label reference. *if* is a convenience for a multistream network.

Records that are selected by the selection stage are processed by the stages between the *if* stage and the first label reference to the stage.

When there is only one label reference to the *if* stage, rejected records are merged with the secondary input stream and passed to the secondary output stream.

When there is a second label reference to the *if* stage, records that are rejected by the selection stage are processed by the stages between the two label references. Records on the secondary input stream are merged with the records on the tertiary input stream and passed to the tertiary output stream.



Type: Gateway.

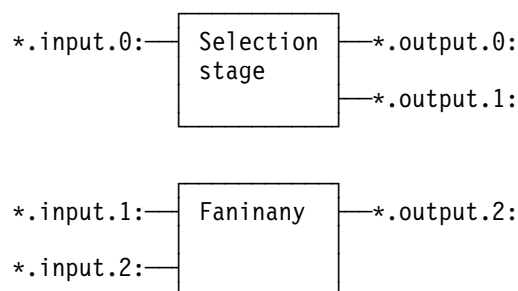
Syntax Description:

word The name of a selection stage.

string The argument string for the selection stage.

Operation: *if* adds a pipeline specification that contains the selection stage and a *faninany* stage to the running pipeline set.

For the full if/then/else configuration, the topology is this:



In the if/then configuration, the secondary output stream from the selection stage is connected directly to the *faninany* stage.

Streams Used: Two streams must be defined; up to three streams may be defined. *if* reads from and writes to all defined streams.

Record Delay: *if* strictly does not delay the record. If the stages between the label references also do not delay the record, the output records will be in the same order as the input records.

Commit Level: *if* starts on commit level -2. *if* does not commit; the selection stage will cause it to commit.

Examples: To upper case records that contain “up” in the first two columns and delete those columns from the upper cased records:

```
'...',
'|if1: if strfind /up/',
'|not chop 2',
'|xlate',
'|if1:',
'|...
```

To mark records that contain the string “abc” and shift the balance of the file three columns to the right:

```
'...',
'|if1: if locate /abc/',
'|insert /-> /',
'|if1:',
'|insert / /',
'|if1:',
'|...
```

Notes:

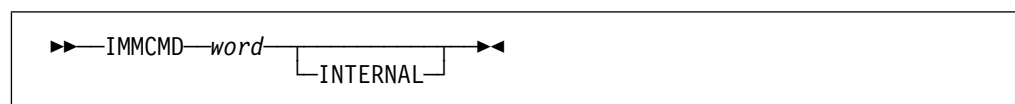
1. *if* does not enforce any particular topology and any particular type of selection stage.
2. Do not use end characters with *if*. In particular, records must be able to flow both into and out of its secondary streams and its tertiary streams.

Return Codes: Unless a message is issued by *if*, the return code is the one from the selection stage.

immcmd—Write the Argument String from Immediate Commands

immcmd sets up a CMS immediate command and waits for it to be issued. When the specified immediate command is issued from the terminal, the argument string (on the command issued) is written to the pipeline.

On z/OS, *CMS Pipelines* implements a scheme like CMS immediate commands. When an attention request is received, the user is prompted for the immediate command to be issued.



Type: Device driver.

Placement: *immcmd* must be a first stage.

Syntax Description: A word is required; it is translated to upper case. A second keyword is optional. When INTERNAL is specified, the immediate command will not of itself cause a wait state, but it will wait while other stages wait for external events (even other *immcmd* stages). Essentially, this boils down to “do not wait for me”.

Commit Level: *immcmd* starts on commit level -1. It sets up an immediate command handler for the specified name and then commits to level 0.

Premature Termination: *immcmd* terminates when it discovers that its output stream is not connected. *immcmd* does not complete normally. *immcmd* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *console*.

Examples: To process (in a service virtual machine) commands issued from the terminal as well as commands sent via SMSG from other users:

```
/* GETCMD REXX */
address command 'CP SET SMSG IUCV'
'callpipe (end ?)',
  '|immcmd cmd',           /* Immediate commands          */
  '|spec ,00000004*, 1.16 1-* next', /* As if SMSG from self      */
  '|f:faninany',          /* Join all                    */
  '|*:',                  /* Pass to output              */
  '?starmsg',             /* Listen for SMSGs           */
  '|f:',                  /* Merge with commands        */
  '?immcmd stop',         /* Stop command                */
  '|pipestop'             /* Force stop                   */
```

Two immediate commands are set up in this pipeline specification. One is for CMD; the output from this *immcmd* stage is transformed into the format for special messages from “*” and merged with special messages received from other users. The output from the second invocation of *immcmd* is passed to *pipestop*, which signals the *immcmd* and *starmsg* stages to terminate.

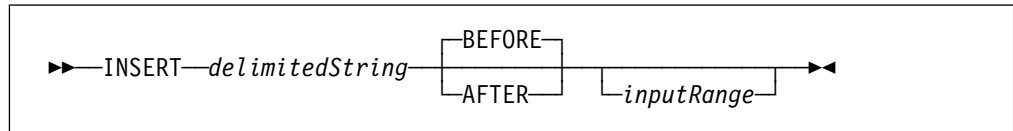
This example also shows a pipeline (the last one) that is not connected to the other pipelines in a set.

Notes:

1. Multiple *immcmd* stages can be used, but the command name should be unique. When more than one stage uses a particular immediate command, it is unspecified which one is started last and receives the commands from CMS.
2. Use *pad 1* to turn null lines into lines with a single blank.
3. *immcmd* may be useful in pipeline specifications containing *delay* or *starmsg* stages, or both.
4. TSO keeps the keyboard locked while a command runs. Hence the need for the attention and the prompt.
5. *immcmd ... INTERNAL* cannot obscure a stall.
6. Use *INTERNAL* only when you do not wish to wait for the user to issue the immediate command, but wish to allow the user to issue the command while the pipeline runs.
You need to take precautions to terminate *immcmd*, for example with *gate*, or avoid a stall when specifying *INTERNAL*.
7. You can find an example server infrastructure at
<http://vm.marist.edu/~pipeline/servus.rexx>

insert—Insert String in Records

insert inserts a string in all input records.



Type: Filter.

Syntax Description:

delimitedString Specify the string to insert.

BEFORE The string is inserted before the *inputRange* (or at the beginning of the record).

AFTER The string is inserted after the *inputRange* (or at the end of the record).

inputRange Specify the position where the string is to be inserted. The default is the entire record.

Operation: When the input range is not present in the record, the range used is the first or the last column of the record, depending on whether the first part of the *inputRange* is relative to the beginning or the end of the record.

Record Delay: *insert* strictly does not delay the record.

Premature Termination: *insert* terminates when it discovers that no output stream is connected.

See Also: *change* and *specs*.

Examples: To append a string to the end of the record:

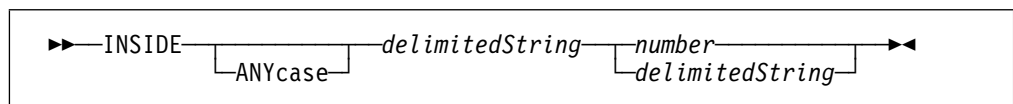
```
pipe literal abc def | split | insert /*/ after | console
▶abc*
▶def*
▶Ready;
```

Notes:

1. *insert* is a convenience; *specs* can perform all the functions that *insert* can perform.

inside—Select Records between Labels

inside selects groups of records whose first record follows a record that begins with a specified string. The end of each group can be specified by a count of records to select, or as a string that must be at the beginning of the first record after the group.



Type: Selection stage.

Syntax Description: A keyword is optional. Two arguments are required. The first one is a delimited string. The second argument is a number or a delimited string. The number must be zero or positive.

Operation: *inside* copies the groups of records that are selected to the primary output stream, or discards them if the primary output stream is not connected. Each group begins with the record after the one that matches the first specified string. When the second argument is a number, the group has as many records as specified (or it extends to end-of-file). When the second argument is a string, the group ends with the record before the next record that matches the second specified string (or at end-of-file).

When ANYCASE is specified, *inside* compares fields without regard to case. By default, case is respected.

inside discards records before, between, and after the selected groups or copies them to the secondary output stream if it is connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *inside* strictly does not delay the record.

Commit Level: *inside* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *inside* terminates when it discovers that no output stream is connected.

Converse Operation: *notin*.

See Also: *between* and *outside*.

Examples: To process the examples in a Script file, discarding the example begin and end tags (assuming these tags are in separate records):

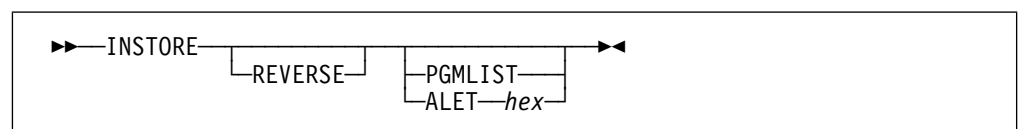
```
...| inside /:xmp./ /:exmp./ |...
```

Notes:

1. With identical string arguments, *inside* differs from *between* in that *inside* does not select the records that match the strings.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
3. *pick* can do what *inside* does and much more.

***instore*—Load the File into a storage Buffer**

instore stores all input records in a data structure and then writes a single descriptor record into the pipeline. A cascade of *instore* REVERSE and *outstore* reverses the order of the records.



Type: Arcane filter.

Syntax Description:

: REVERSE Store records in the reverse order they are read.
:
: PGMLIST The output record is a descriptor list, as needed for a REXX program in
:
: storage.
:
: ALET The file is buffered in the specified data space.

: PGMLIST and ALET are mutually exclusive.

: **Operation:** The input is read into a buffer in virtual storage or the specified data space.
:
: At end-of-file, an output record is produced describing the file.

When the keyword PGMLIST is specified, the descriptor list is built as specified for REXX programs in storage; the descriptor list is written to the output.

When the keyword PGMLIST is omitted, the file is stored in a chained list of records. Each record has an eight byte prefix consisting of a pointer to the next record and a fullword length.

: The output record is valid only until it is consumed; the buffers are released after that.

: When ALET is specified, the data space must have been created by ADRSPACE CREATE
:
: INITIALISE or equivalent. It must have the storage key under which *CMS Pipelines*
:
: executes, which is X'EO' on CMS and X'80' on z/OS. *instore* locks the data space with a
:
: key identifying itself to ensure exclusive access; the lock is released when *instore* termi-
:
: nates, thus making the data space available for other use.

: **Output Record Format:** When the keyword PGMLIST is omitted, the format of the record
:
: written is defined in the STORBUF member of FPLGPI MACLIB and in the built-in structure
:
: fplstorbuf.

Record Delay: *instore* delays all records until end-of-file.

Commit Level: *instore* starts on commit level -2. When ALET is specified, it verifies the integrity of the data space and then commits to level 0.

Converse Operation: *outstore*.

See Also: *buffer*.

Examples: To reverse the order of the lines in a file:

....| instore reverse | outstore |...

To send a file to two destinations:


```

/* Process an in-store file */
'addpipe *: | instore | *.input:'
'peekto file_descriptor'
address command 'CP TAG DEV 00D GDLVM7 PIPER'
'callpipe var file_descriptor | outstore | punch'
address command 'CP CLOSE 00D NAME for piper'
address command 'CP TAG DEV 00D CPHVM1 JOHN'
'callpipe var file_descriptor | outstore | punch'
address command 'CP CLOSE 00D NAME for john'
'readto'

```

When processing the output from *instore* in a REXX program, use the PEEKTO pipeline command to read the line describing the file. Issue the READTO pipeline command without operands to consume the descriptor record after the file has been processed.

As long as the descriptor record has not been consumed, *instore* holds the entire input data. When multiple copies of the descriptor record are passed to *outstore* (for instance with *dup*) then *outstore* will produce multiple copies of the input data. Compare the two examples with and without the *instore* and *outstore* pair.

```

pipe literal I must not talk in class | spill 13 | instore | dup | ...
... outstore | console

```

```

▶I must not
▶talk in class
▶I must not
▶talk in class
▶Ready;

```

```

pipe literal I must not talk in class | spill 13 | dup | console

```

```

▶I must not
▶I must not
▶talk in class
▶talk in class
▶Ready;

```

Notes:

1. No built-in program can process the output from *instore* PGMLIST. For all practical purposes *outstore* is the only way to process the output from *instore* without the PGMLIST option. In either case, do not attempt to process the file token with filter stages.
2. The output record describes the file. The output record must be obtained with a locate mode call and completely processed before it is consumed; the buffer is returned to the operating system by *instore* immediately after the record is consumed.
3. The file may be extracted by passing the record to *outstore* or, when ALET is specified, it may be extracted in a different virtual machine that has obtained access to the data space, as long as the output record is not consumed in the creating virtual machine. The extracting virtual machine will typically use a different ALET from the creating virtual machine.
4. Using the ALET operand may offer virtual storage constraint release, as well as data transfer between virtual machines.

ip2socka—Build sockaddr_in Structure

ip2socka converts a readable port number and IP address to sixteen bytes socket address structure, which can be used with *udp*.

►►—IP2SOCKA—◄◄

Type: Filter.

Input Record Format: Three blank-delimited words:

1. The literal constant “AF_INET”.
2. The port number, which is in the range 0 to 65535, inclusive.
3. The IP address in dotted-decimal notation or the host name and domain name.

Output Record Format: A structure of sixteen bytes. Binary numbers are stored in the network byte order, that is, with the most significant bit leftmost.

Pos	Len	Description
1	2	The short unsigned number 2, which specifies that the addressing family is AF_INET.
3	2	The short unsigned port number.
5	4	The unsigned IP address.
9	8	Reserved. Binary zeros

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *ip2socka* does not delay the record.

Premature Termination: *ip2socka* terminates when it discovers that its output stream is not connected.

Converse Operation: *socka2ip*.

Examples: To convert an address to internal format and convert this structure to printable form:

```
pipe literal af_inet 17 1.2.3.4 | ip2socka | spec 1-* c2x 1 | console
►00020011010203040000000000000000
►Ready;
pipe literal af_inet 17 0300.2.3.4 | ip2socka | spec 1-* c2x 1 | console
►00020011C00203040000000000000000
►Ready;
pipe literal af_inet 17 0xc0.2.3.4 | ip2socka | spec 1-* c2x 1 | console
►00020011C00203040000000000000000
►Ready;
```

Notes:

1. On CMS, you can specify a host name or a host name followed by a domain. *CMS Pipelines* calls *RXSOCKET* to do the actual name resolution. As a consequence, the

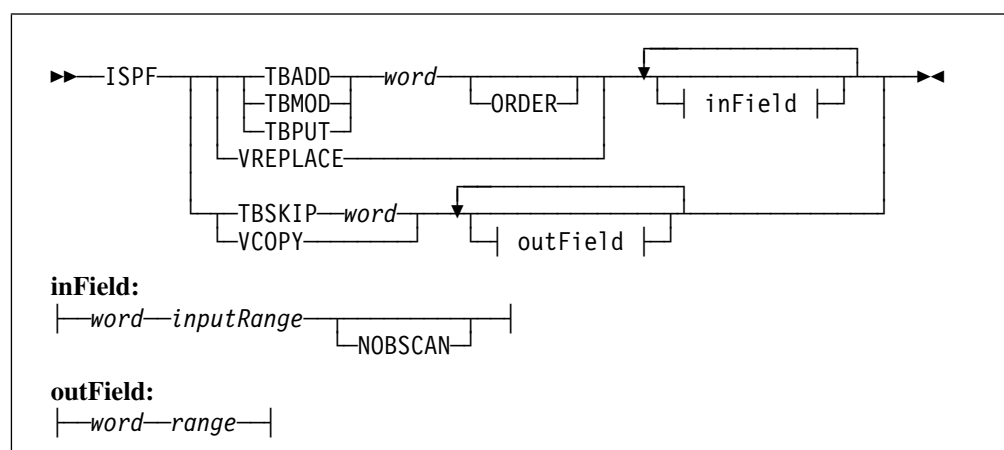
name is resolved using RXSOCKET rules. This implies that the file TCPIP DATA must be available and must point to the name server. RXSOCKET (unlike *CMS Pipelines*) uses the server virtual machine specified in TCPIP DATA.

ispf—Access ISPF Tables

ispf TBSKIP reads rows from an ISPF table into the pipeline. *ispf* TBADD adds rows containing data from the pipeline; *ispf* TBMOD and *ispf* TBPOT modify rows to insert data from the pipeline. The table must have been created and opened by some other means (for instance *subcom* ISPEXEC TBCREATE) before *ispf* can access it.

ispf VCOPY reads the contents of ISPF variables into the pipeline. *ispf* VREPLACE stores the contents of input records into ISPF variables.

See Chapter 12, “Using *CMS Pipelines* with Interactive System Productivity Facility” on page 145 for task-oriented information.



Type: Device driver.

Placement: *ispf* TBSKIP must be a first stage. With other operands, *ispf* must not be a first stage.

Syntax Description: The name of an ISPF service for table operation (TBADD, TBMOD, TBPOT, or TBSKIP) is followed by an optional list of field definitions. These specify the location of variables in the input or output record. The table name and the variable names are translated to upper case. The keyword ORDER with *ispf* TBADD and TBMOD indicates that rows are inserted in some predefined order. Refer to TBADD ORDER in *Dialog Management Services and Examples*, SC34-4010.

The operands VCOPY and VREPLACE support access to ISPF function pool variables without issuing a table request. At least one field must be specified with these operands.

The field definitions specify the positions of fields in the input and output records. For input records, specify the field name and an input range; use the optional keyword NOBSCAN to retain trailing blanks. For output records, specify the name and a column range. For compatibility with the past, the keyword CHAR is ignored if it is specified after the name of the field.

Operation: For TBADD, TBMOD, and TBPOT, variables in the function pool are set (using the VREPLACE service) to the contents of the fields in the input record, stripped of trailing blanks unless NOBSCAN is specified. The requested ISPF table service is then invoked to copy the values from the function pool into the table. The input record is then copied to the output (if connected).

For TBSKIP, the ISPF service TBSKIP is called and an output record is written containing the contents of the specified variables (obtained by the VCOPY service); a null record is written when no variables are specified. This process is repeated until ISPF sets return code 8 (end of table).

ispf VCOPY discards input records. For each input record, it copies the specified variables from the function pool and stores them in an output record. The output record is written before the input record is consumed.

ispf VREPLACE stores the contents of the specified fields into function pool variables. It copies the input record to the output (if connected) before consuming it.

Input Record Format: As defined by the field definitions.

Output Record Format: When *ispf* is first in a pipeline (*ispf* TBSKIP), the output record contains data from variables, as defined by field definitions in the argument list. Positions between fields are blank.

Streams Used: Records are read from the primary input stream and written to the primary output stream. When *ispf* is not first in the pipeline and VCOPY is not specified, the input record is copied to the output after the ISPF service has been performed.

Record Delay: *ispf* strictly does not delay the record.

Commit Level: *ispf* starts on commit level -2000000000. It processes the argument string, allocates a buffer to hold the data in the fields, and then commits to 0.

Premature Termination: *ispf* terminates on most nonzero ISPF return codes. *ispf* with the option TBSKIP or VCOPY terminates when it discovers that its output stream is not connected.

Examples: To store the names of the files on the CMS system disk in a table:

```
/* ISPF test cases.                                     */
signal on novalue
signal on error
trace error

address ispexec,
  'TBCREATE LISTFILE NAMES(FN FT FM REST) WRITE REPLACE'

address command
'PIPE (end \ name INSPIPE )',
  'cms listfile * * s',
  '| ispf tbadd LISTFILE',
  '| FN 1.8 FT 10.8 FM 19.2 REST 21-80',
  '| count lines',
  '| var lines'

say lines 'stored.'
```

```
'FILEDEF $$ DISK $$DUMMY $$TABLE A'
```

```
address ispexec,  
  'TBCLOSE LISTFILE LIBRARY($$)'
```

```
error: exit RC
```

Notes:

1. To read all rows of a table, position the cursor for the table before the first row prior to using *ispf* TBSKIP (for example with *subcom* ISPEXEC TBTOP).
2. Never address a command to ISPLINK; results are unpredictable.
3. Return code 20 is reflected by ISPEXEC in a REXX program when ISPF does not recognise the service requested. This can happen, for example, when a PIPE command is addressed to ISPEXEC rather than to CMS. (This is not a return code from the *ispf* stage itself.)
4. When more than one field for *ispf* TBSKIP refers to the same range, it is unspecified which field is stored in the output record.
5. Extension variables are not supported directly. Use *ispf* VREPLACE to set variables to the contents of a record and then issue TBADD to the ISPEXEC subcommand environment, specifying the extension variables.

jeremy—Write Pipeline Status to the Pipeline

jeremy writes the status of all stages in the current pipeline set to its output stream whenever an input record becomes available.

jeremy is useful when debugging a complex multistream pipeline in which data have ceased to flow in some pipeline segments.

►►—JEREMY—◄◄

Type: Service program.

Placement: *jeremy* must not be a first stage.

Operation: If the secondary output stream is defined, *jeremy* passes the input line to the primary output stream after it has written the pipeline status to the secondary output stream; if there is no secondary output stream defined, *jeremy* discards the input record after it has written the pipeline status to the primary output stream.

Output Record Format: The output format is unspecified.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *jeremy* does not delay the last record written for an input record.

Commit Level: *jeremy* starts on commit level -2. It verifies that the primary input stream is the only connected input stream and then commits to level 0.

join

Premature Termination: *jeremy* terminates when it discovers that any of its output streams is not connected.

Examples:

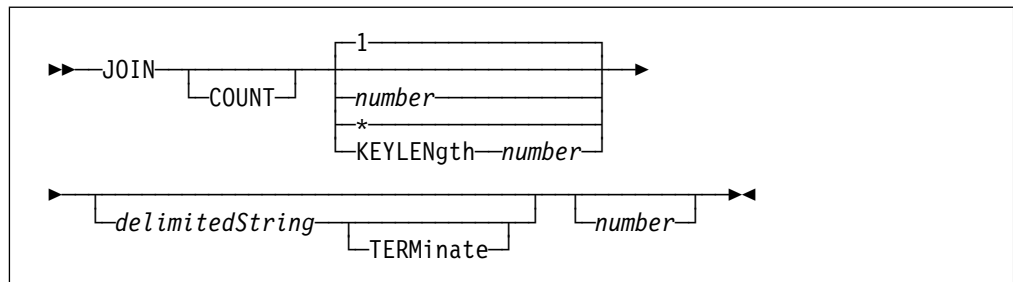
```
pipe literal | jeremy | console
▶5785-RAC CMS Pipelines level 1.1.12 sublevel 17 ("0011"x)
▶ From z/VM 7.1.0
▶
▶Pipeline specification 1          commit 0
▶ Pipeline 1
▶ literal                          wait.out.
▶ record on output 0:              ""
▶ jeremy                            ready.
▶ console                          wait.locate.
▶Ready;
```

Notes:

1. When the configuration variable `STALLACTION` is set to `JEREMY`, *CMS Pipelines* invokes *jeremy* when the pipeline is stalled, to note the state of the pipeline set.
2. The output of *jeremy* is not buffered. The stages that process the output of *jeremy* must not change the pipeline topology.

join—Join Records

join puts input records together into one output record, optionally inserting a specified string between joined records. All input lines are joined into one output record when an asterisk is specified. The maximum length of an output record can also be specified.



Type: Filter.

Syntax Description: A keyword, a number or an asterisk, a delimited string, and a number are the optional arguments.

COUNT Prefix each record written to the primary output stream with the count of the contributing input records. The number is 10 characters, aligned to the right.

number Unless `KEYLENGTH` is specified, the first number specifies how many lines are appended to the first one in a set; it can be zero or more. The default is to join pairs of lines unless the secondary input stream is defined, in which case the default is infinity.

KEYLENGTH Join lines that contain the same leading string. The number specifies the length of key string.

delimitedString Insert the contents of the specified string between joined records.

TERMINATE Append the string also after the last record in a set.

number The second number specifies the maximum output record length exclusive of the count prefix, if any. To be recognised as the maximum record length, the number cannot be the first word of the arguments (because it would then specify the number of lines to append rather than the maximum record length).

Operation: When a maximum record length is specified, records are joined as defined by the other arguments until an input record would cause more data than specified to be loaded into the output buffer. The contents of the output buffer (if any) are flushed to the output and a new set of records is processed. An input record with a length that is equal to or greater than the maximum output record length is written unchanged (that is, it is not truncated).

When KEYLENGTH is specified, records are joined as long as the keys are equal (and the maximum length has not been exceeded). The key is discarded from records 2 to n in a set of joined lines.

When the secondary input stream is defined, the buffer is flushed whenever a record arrives on it. The record on the secondary input stream is then discarded. Thus, *join* does not delay the output from a record on the secondary input stream.

Streams Used: Secondary streams may be defined. Records are written to the primary output stream; no other output stream may be connected. The primary input stream is shorted to the primary output stream if the number is zero.

Record Delay: When both KEYLENGTH and a maximum record length are omitted, *join* does not delay the last record written for an input record.

When KEYLENGTH or a maximum record length (or both) is specified, the output record is delayed to the record following the last one joined.

Commit Level: *join* starts on commit level -2.

Premature Termination: *join* terminates when it discovers that its primary output stream is not connected. End-of-file on the primary input stream is ignored when the secondary input stream is defined. *join* terminates when it discovers that its output stream is not connected.

See Also: *spill* and *split*.

Examples: To join two records with an asterisk between them:

```
pipe literal a b c d e | split | join /*/ | console
▶a*b
▶c*d
▶e
▶Ready;
```

To flow text into 10-character columns:

joincont

```
pipe literal This is a short sentence. | split | join * / / 10 | console
▶This is a
▶short
▶sentence.
▶Ready;
```

To capitalise the first letter of all words while retaining the original record structure:

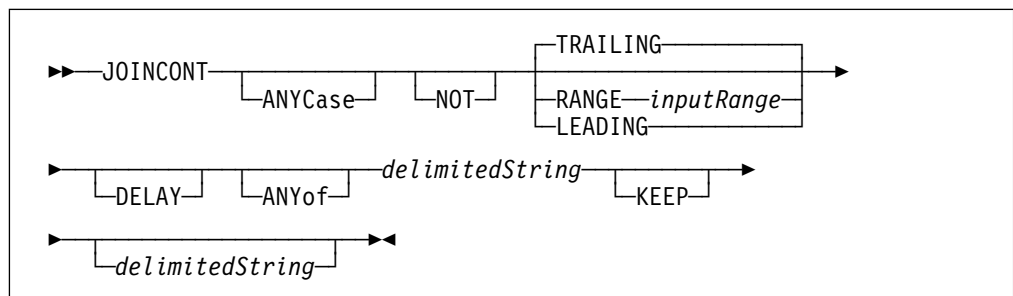
```
'(end ?) ... ',
'| o: fanout ',
'| split after blank ',          /* Retain multiple blanks */
'| xlate 1 ',
'| j: join ',                    /* Rebuild record */
'| ... ',
'? o: ',
'| j:'
```

Notes:

1. The secondary input stream is intended for solutions where an input record is chopped up into bits that need to be processed separately and then joined. You would *fanout* a copy of the original record and cause that to flush *join*'s buffer. Used in this way, *join* does not delay the record relative to the stage that produces the record on the secondary input stream to *join*.

joincont—Join Continuation Lines

joincont joins records that are marked with a continuation string.



Type: Filter.

Syntax Description:

ANYCASE	Case is ignored when comparing strings.
NOT	The absence of the specified string will cause records to be joined.
TRAILING	The continuation string is at the end of the record being continued. When the continuation string is present, the following record will be appended to the record that contains the string. This is the default.
RANGE	Specify an input range to examine. When the continuation string is present, the following record will be appended to the record that contains the string. RANGE implies KEEP.
LEADING	The continuation string is at the beginning of the record following the one being continued. That is, when the continuation string is present, the record will be appended to the previous record.

:	DELAY	DELAY is used with TRAILING or RANGE. DELAY is ignored if it is specified with LEADING. When DELAY is specified, the last input record for a particular output record is consumed before the record is written. This may save a <i>copy</i> stage.
.		
.		
.		
.	ANYOF	The following delimited string enumerates characters to be tested as continuation. To determine if continuation exists, a single character in the input record is compared against the characters in the string. A continuation exists when any character of the string matches the character. The default compares the string against a leading or trailing string of the same length.
.	<i>delimitedString</i>	The first delimited string specifies the continuation string. When ANYOF is specified, the string enumerates a set of characters; when ANYOF is omitted, the delimited string represents a normal string. This string is deleted from the output record unless NOT or KEEP is specified.
.	KEEP	The continuation string is retained. The default is to delete the continuation string unless RANGE is specified. RANGE implies KEEP.
.	<i>delimitedString</i>	The second delimited string specifies the string to be inserted between the two records in the output record.

Operation: When ANYOF is specified, the continuation criterion is whether the last character of a record (or the first character of the following one; or the contents of the specified input range) is present in the string representing an enumerated set of characters. That is, the character must compare equal to at least one of the set. When ANYOF is omitted, the string is compared character for character with the end or beginning of a record; all character positions must compare equal. When NOT is specified, the criterion is inverted; the absence of the string or character defines continuation.

When the keyword RANGE is specified, the input range is inspected for continuation; when the keyword TRAILING is specified or defaulted, the trailing part of each input record is inspected for continuation. When continuation is not indicated, the record is passed unmodified to the output. When continuation is indicated, the record is loaded into a buffer; the string or character is deleted if all of RANGE, NOT, and KEEP are omitted. The second delimited string is appended to the contents of the buffer; the next input record is read and appended to the buffer. The new record is then inspected for continuation. This process continues as long as the record appended to the buffer triggers continuation. The contents of the buffer are written to the output at the end of a run of continued records.

When LEADING is specified, an input record is read into a buffer and the leading string or character of the next input record is inspected for continuation. When no continuation is indicated, the contents of the buffer are written to the output and the process repeats by loading the second record into the buffer. If continuation is indicated, the second string is appended to the contents of the buffer followed by the remainder of the input record. This process continues until a record is read that does not contain the specified leading string. When this happens, the contents of the buffer are written to the output, the input record is loaded into the buffer and the process is repeated.

Record Delay: *joincont* TRAILING and *joincont* RANGE do not delay records that are not continued. When DELAY is omitted they do not delay the last record of a set of continuation records. *joincont* LEADING delays records that are not continued by one record; it delays the last record of a run of continued records by one record.

Premature Termination: *joincont* terminates when it discovers that its output stream is not connected.

See Also: *asmcont*, *join*, and *deblock*.

Examples: To join records that have been split in an RFC 822 header:

```
/* Join headers: */
'PIPE (end ? name JOINCONT)',
  '| ... ',
  '| xlate *-* 05 blank',          /* Tabs to blanks          */
  '| joincont leading / / keep',
  '| ...
```

This example would be simplistic for splicing paragraphs of text because it keeps multiple leading blanks, which will appear in the spliced record. Also, it does not show how to terminate processing at the blank line that ends the header part of the message.

Notes:

1. Note that ANYOF has precedence as far as abbreviations are concerned. Specify at least four characters of ANYCASE.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
3. *joincont* not range 1 /x/ joins records until and including the next one where column 1 contains "x".

juxtapose—Preface Record with Marker

juxtapose reads input records from the primary input stream or the secondary or higher-numbered input streams as they become available. *juxtapose* writes a record to the primary output stream only when it reads a record from its secondary or higher-numbered input streams. This record contains the last record read from the primary input stream with the record from the secondary or higher-numbered input streams appended to it.



Type: Gateway.

Syntax Description:

COUNT Prefix each record written to the primary output stream with a count that tracks the number of records from the secondary or higher-numbered input streams that are matched with a particular record from the primary input stream. The count is a 10 digit number, aligned to the right.

Operation: Each record from the primary input stream is stored in a buffer, replacing the previous contents of this buffer. The input record is then consumed.

When a record is read from the secondary or higher-numbered input streams, it is appended to the record currently in the buffer; the combined record is then written to the primary output stream. If **COUNT** is specified, the combined output record is prefixed with the count of records that have been appended to the current contents of the buffer. This count starts as one.

When a record is available on the primary input stream and no record was read from the secondary or higher-numbered input streams while the record resided in the buffer, the contents of the buffer are written to the secondary output stream before the next record from the primary input stream is read into the buffer. The keyword `COUNT` has no effect on this operation; only the contents of the original record are written.

Streams Used: *juxtapose* reads all input streams; it writes only to the primary output stream and the secondary output stream.

Record Delay: *juxtapose* does not delay output records on the primary output stream. These records are derived from the records read on the secondary or higher-numbered input streams. Records on the secondary output stream are delayed by one record relative to the primary input stream.

Commit Level: *juxtapose* starts on commit level -2. It verifies that the tertiary output stream and higher-numbered output streams are unconnected and then commits to level 0.

Premature Termination: *juxtapose* terminates when it discovers that no output stream is connected.

See Also: *predselect*.

Examples: To construct a very simple file scanning routine, the following scans a set of files and shows the matching lines prefixed by the name, type, and mode of files:

```
'PIPE (end ? name JUXTAPOS.STAGE:51) ',
  '?cms listfile * EXEC',          /* Get list of files          */
  '|o:fanout',                    /* Make two copies           */
  '|pad 25',                       /* Some space to align       */
  '|j:juxtapose',                 /* Prefix to contents of file */
  '|i:faninany',                 /* Combine both streams      */
  '|...',                         /* Do whatever               */
  '|?o:',                         /* The file ids              */
  '|getfiles',                   /* Read contents             */
  '|locate anycase ,pipe,',      /* Find lines with this string */
  '|j:',                          /* Go merge with name        */
  '|insert , *no match*, after', /* Mark file as no match     */
  '|i:'                           /* Feed into the same stream  */
```

This example deserves scrutiny. Several conditions must be satisfied for it to work correctly; that is, reliably prefix the name of the file to each record of the file:

- The two input streams to *juxtapose* are derived from a common source; in this case, *fanout*.
- *pad* does not delay the record.
- *getfiles* writes the contents of the file to its output before it consumes the corresponding input record.
- *juxtapose* writes an unmatched name to the secondary output stream before it consumes the input record, so *faninany* gets the records in sequence.

Thus, even though the order of dispatching is undefined, the dispatcher is not given any leeway; sooner or later it must produce the records to *juxtapose*. And at any one time, the dispatcher can produce a record on only one of the input streams.

In contrast, this pipeline specification is indeterminate:

```

    pipe (end ?) literal abc|j:juxtapose|cons?literal def ghi|split|j:
▶abcdef
▶abcghi
▶Ready;

```

It might produce two records containing abcdef and abcghi or it might produce only def and ghi; it depends on which of the two *literal* stages produces a record first.

The following examples show that in the current implementation *cp* seems to produce a record after *literal* does; but this could change in the future.

```

    pipe (end ?) literal abc |j:juxtapose|cons?cp query time|j:
▶abc TIME IS 14:50:34 EDT WEDNESDAY 04/29/20
▶abc CONNECT= 99:59:59 VIRTCPU= 069:09.28 TOTCPU= 069:44.69
▶Ready;

```

The following shows that a small change to the pipeline specification can change the dispatching order (the source for the two streams was swapped).

```

    pipe (end ?) cp query time|j:juxtapose|cons?literal abc def|split|j:
▶abc
▶def
▶Ready;

```

To prefix the first byte (the operation code) of the input of a *fullscr* stage to the corresponding *fullscr* output records:

```

/* Retain opcode */
'PIPE (end ?)',
  | ...
  |o: fanout ',          /* copy input to fullscr      */
  | chop 1 ',          /* keep only the opcode      */
  |j:juxtapose ',      /* preface output with opcode */
  | ...                /* process response here     */
  '?o:',
  | fullscr',          /* write input to screen     */
  |j:                 /* fullscr output to juxtapose */

```

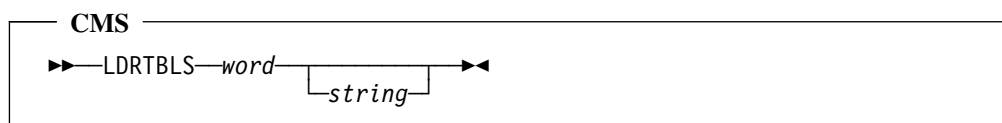
To ensure that the correct prefix is available at the time it is needed, we connect the primary output stream from *fanout* to the primary input stream of *juxtapose*. Thus, the record will be available to *juxtapose* before it is available to *fullscr*, and therefore before *fullscr* generates an output record.

ldrtbls—Resolve a Name from the CMS Loader Tables

ldrtbls resolves an entry point in the loader tables. The entry point can be:

- An executable machine instruction (not X'00'), which is then invoked as a stage.
- A program descriptor, which describes the stage to run.
- A pipeline command, which is run as a subroutine pipeline.
- An entry point table, from which the first word of the argument string is resolved.
- A look up routine, which resolves the first word of the argument string.

ldrtbls is often used to test a compiled REXX program or an Assembler program before it is generated into a filter package.



Type: Arcane look up routine.

Syntax Description: Leading blanks are ignored; trailing blanks are significant. A word is required; additional arguments are allowed. The entry point specified by the first word is looked up in the CMS loader tables. If no entry point is found with the name as specified, it is translated to upper case and the loader tables are searched again.

Operation: The optional string is passed to the program as the argument string.

Record Delay: *ldrtbls* does not read or write records. The delay depends on the program being run.

Examples: To test a compiled REXX filter before it is put in a filter package:

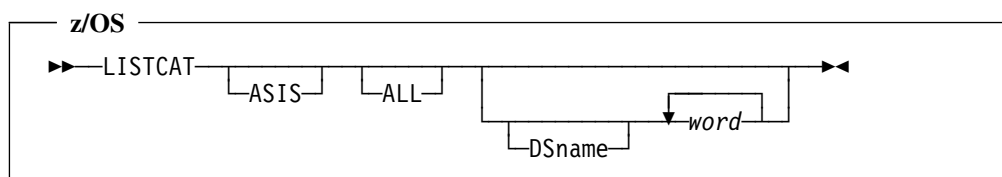
```
rexxcomp myfilter rexx ( object
load myfilter
pipe ldrtbls myfilter|...
```

Notes:

1. *ldrtbls* is useful to test a new version of a filter loaded in the user area while still retaining the production version for normal use.
2. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

listcat—Obtain Data Set Names

listcat writes a list of data set names into the pipeline. The list is qualified by a specified string or the current prefix, or both.



Type: Device driver.

Syntax Description:

ASIS	Use data set names as written; do not translate to upper case. The default is to translate data set names to upper case.
ALL	Write all entries supplied to the pipeline, prefixed by a one character code. By default, only data set names and VSAM cluster names are written.

listdsi

DSNAME A list of data set qualifiers follows. DSNAME is assumed in front of an option that is not recognised.

Operation: Names are listed for each word in the DSNAME list and each word in the input records. When the word does not begin with a quote and a prefix is set, the prefix and a period are added to the front of the word specified. When the word begins with a quote it is used as it is.

Output Record Format: When ALL is specified, each output record contains a character code in the first column:

A	Non-VSAM data set.
B	Generation data group.
C	Cluster.
G	Alternate index.
H	Generation data set.
L	Tape volume catalog library entry.
R	VSAM path.
U	User catalog connector entry.
W	Tape volume catalog volume entry.
X	Alias.

The name follows from column 2.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *listcat* does not delay the last record written for an input record.

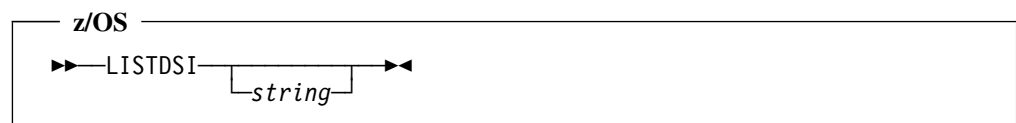
See Also: *listdsi* and *sysdsn*.

Examples: To list all data sets that begin with the letter T (the prefix is DPJOHN):

```
pipe listcat tso.f | term
▶DPJOHN.TSO.F
▶DPJOHN.TSO.FB
▶DPJOHN.TSO.FF
▶DPJOHN.TSO.FILTERS
▶READY
```

listdsi—Obtain Information about Data Sets

listdsi calls the REXX function `listdsi()` and writes the contents of the the variables that describe the data set to the primary output stream. It then writes the function result (the return code) to the secondary output stream (if it is defined).



Type: Device driver.

Syntax Description: If an argument string is specified, it is processed before *listdsi* reads input records, as if it were an input record.

Operation: The argument string (if it is not blank) and each non-blank input record are passed to the `listdsi()` function without inspection or modification. When the return code is less than 16, the variables shown below are obtained from the REXX environment and written to the output stream if they are defined. Directory information (ADIRBLK, UDIRBLK, and MEMBERS) is written only if the return code is 0. The variables are written in the order shown, column by column. When the return code is 16, only the last three variables are obtained.

DSNAME	RECFM	ALLOC	UNITS	EXDATE	TRKSCYL	MEMBERS
VOLUME	LRECL	USED	EXTENTS	PASSWORD	BLKSTRK	REASON
UNIT	BLKSIZE	PRIMARY	CREATE	RACFA	ADIRBLK	MSGLV1
DSORG	KEYLEN	SECONDS	REFDATE	UPDATED	UDIRBLK	MSGLV2

Output Record Format: The lines that are written to the primary output stream contain a variable name and its value in a format that is compatible with *varset*.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded.

Record Delay: *listdsi* writes all output for an input record before consuming the input record.

Commit Level: *listdsi* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: When the secondary output stream is not defined, *listdsi* terminates when it discovers that its output stream is not connected. When the secondary output stream is defined, *listdsi* terminates when it discovers that its secondary output stream is not connected; it ignores end-of-file on the primary output stream.

See Also: *sysdsn* and *state*.

Examples:

```

pipe listdsi sys1.proclib | take 3 | terminal
▶=SYSREASON=0005
▶READY
pipe listdsi 'sys1.proclib' | take 3 | terminal
▶=SYSDSNAME=SYS1.PROCLIB
▶=SYSVOLUME=CCAR02
▶=SYSUNIT=3380
▶READY
pipe (end ?) 1: listdsi tso.exec | take 3 | terminal ? 1: | terminal
▶=SYSDSNAME=DPJOHN.TSO.EXEC
▶=SYSVOLUME=FS8E70
▶=SYSUNIT=3380
▶0
▶READY

```

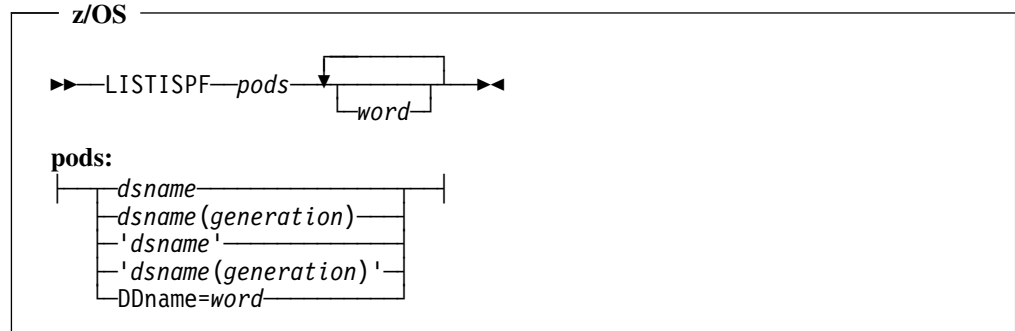
Notes:

1. Data set names follow the TSO conventions. Enclose a name that is fully qualified in single quotes. The prefix is applied to data set names that are not enclosed in quotes.

Return Codes: The return code is 0, irrespective of the return codes from LISTDSI.

listispf—Read Directory of a Partitioned Data Set into the Pipeline

listispf reads the directory of a partitioned data set and writes a line for each member, formatting the information that ISPF adds to the directory. Only the member name is written for members that have not been stored by ISPF.



Type: Device driver.

Placement: *listispf* must be a first stage.

Syntax Description: Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the `DSNAME` without quotes to have the prefix, if any, applied. Append parentheses containing a signed number to specify a relative generation of a data set that is a member of a generation data group. To read the directory of an already allocated data set, specify the keyword `DDNAME=` followed by the `DDNAME` already allocated. The minimum abbreviation is `DD=`.

Specify a list of one or more member names to restrict the information written to these members; the default is to write a line for each member of the data set. It opens the DCB and then commits to level 0.

Operation: If no member names are specified, *listispf* writes a record (in alphabetical order) for each member for the data set. If one or more members are specified, *listispf* writes a record for each of those members in the order they are specified.

Output Record Format:

Pos	Len	Description
1	8	Member name.
11	5	Version and modification.
17	8	Member creation date in ISO format (yyyymmdd).
26	8	Member last update date in ISO format (yyyymmdd).
35	8	Time on a 24-hour clock (hh:mm:ss).
44	5	Size of member.
50	5	Original size of member.
56	5	Number of modified lines.
62	8	User who stored the member last.

Commit Level: *listispf* starts on commit level -1.

Premature Termination: *listispf* terminates when it discovers that its output stream is not connected.

See Also: *listpds*.

Examples: To list the directory of the first data set allocated to SYSEXEC.

```

pipe listispf dd=sysexec | console
▶ALLOCFPL
▶EPOP
▶RT
▶SC      01.00 19921218 22:08   136   136     0 DPJOHN
▶TFT
▶TISP    01.03 19921215 14:57    14    10     0 PIPER
▶TISPF

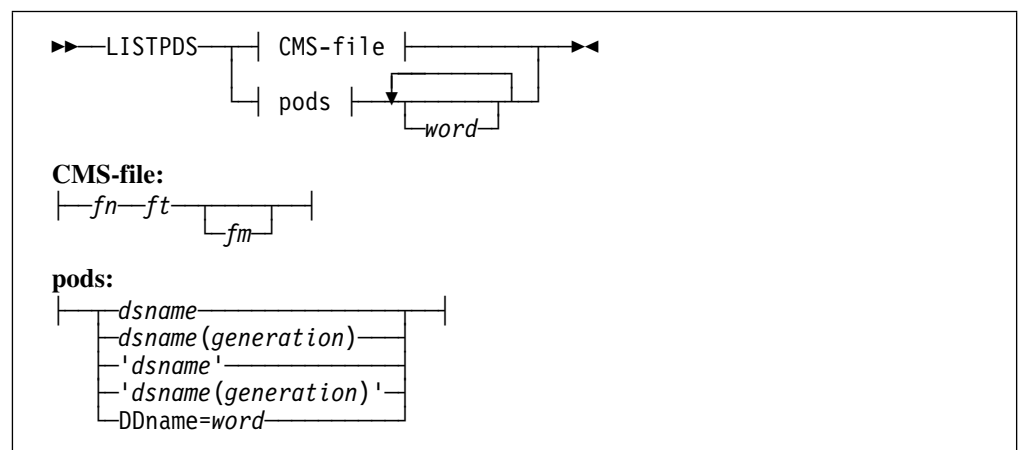
```

Notes:

1. *pdslisti* is a synonym for *listispf*.
2. Refer to the usage notes for *listpds* for information on how to list the members of all data sets that are allocated to a particular DDNAME.
3. The flag for the member being stowed by the Software Configuration and Library Manager is not processed.
4. Members written by *TSO Pipelines* 1.1.9 sublevel 40 (X'0028') and previous versions are stored with the seconds as 01.

listpds—Read Directory of a Partitioned Data Set into the Pipeline

listpds reads the directory of a partitioned data set and writes a line for each member. The first eight bytes contain the member name; the remainder of the output record depends on the particular operating environment and the type of data set.



Type: Device driver.

Placement: *listpds* must be a first stage.

listpds

Syntax Description: CMS: Specify as blank-delimited words the file name and the file type of the file to be read. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried.

z/OS: Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the DSNNAME without quotes to have the prefix, if any, applied. Append parentheses containing a signed number to specify a relative generation of a data set that is a member of a generation data group.

To read the directory of an already allocated data set, specify the keyword DDNAME= followed by the DDNAME already allocated. The minimum abbreviation is DD=.

Specify a list of one or more member names to restrict the information written to these members; the default is to write a line for each member of the data set.

Operation: On CMS, a record is written for each member of the library. Note that libraries can have more than one member of a particular name.

On z/OS, *listpds* can write information about selected members. If no member names are specified, *listpds* writes a record (in alphabetical order) for each member for the data set. If one or more members are specified, *listpds* writes a record for each of those members in the order they are specified.

Output Record Format:

Pos	Len	Description
1	8	Member name.
9	v	Additional information.

Commit Level: *listpds* starts on commit level -2. On z/OS, *listpds* starts on commit level -2000000000. It opens the DCB and then commits to level 0.

Premature Termination: *listpds* terminates when it discovers that its output stream is not connected.

See Also: *listispf*, *members*, and <.

Examples: To list the members of a z/OS load library:

```
pipe listpds tso.load | chop 8 | console
►PIPE
►READY
```

To list the members of a CMS macro library:

```

pipe listpds hcpgpi maclib | chop 8 | pad 10 | snake 7 | console
▶HCPCALL DF8PARAM HCPMWTBK IPARML VRDCBLOK HCPATTRB STHYI
▶ADSR HCPARSPL HCPPPLBK IPARMLX ADRSPACE HCPATTRQ VMUDQ
▶CCED HCPBELBK HCPSBIOP IRAQVS ALSERV HCPREGCK HCPCSIBK
▶CPED HCPBIOPL HCPSGIOP SDMDEVTP APPCVM IUCV HCPINFBK
▶DD4PARAM0 HCPBPLBK HCPSGBK SFBLOK DEFWORKA MAPMDISK HCPMDLAT
▶DD8PARAM0 HCPDE4PL HCPSRBK SPLINK DIAG PFAULT $MLB$
▶D88PARAM0 HCPD290P HCPSXIBK VMCMHDR HCPATTGB REFPAGE
▶DEVTPES HCPMRQBK HCPSXOBK VMCPARM HCPATTOP SETNXTID
▶Ready;

```

Notes:

1. On z/OS, *pdsdirect* and *pdslist* are synonyms for *listpds*.
2. On CMS, *listpds* supports only simulated libraries that have fixed 80-byte records. It does not support a partitioned data set on an OS volume.
3. On z/OS, *listpds* reads the directory of the first data set in a concatenation when the operand specifies a DDNAME. To read the directories of all data sets in a concatenation:

```

/* Read all directories from a concatenation. */
Signal on novalue
parse upper arg ddname .

'callpipe (end ? name ALLDIRS)',
'|tso lista status', /* List allocations */
'|drop 1', /* Remove title */
'|nfind TERM', /* Discard TERMFILLES */
'|spec w1 15 read 3.8 1', /* Prefix DDNAME to DSNAME */
'|frlabel' ddname, /* Select the DDNAME */
'|t:take 1', /* Take the first one */
'|i:fanin', /* And the concatenations */
"spec /callpipe listpds '/ 1 15-* next '/'|*:/ next",
'|pipcmd', /* Issue subroutine to read one */
'|*:', /* Write result */
'?t:', /* Rest of datasets */
'|whilelabel ', /* Get all concatenations */
'|i:' /* And process... */

exit RC

```

literal—Write the Argument String

literal writes its argument string into the pipeline and then passes records on the input to the output stream.



Type: Device driver.

Syntax Description: The string starts after exactly one blank character. Leading and trailing blanks are significant.

Operation: *literal* writes a null record when the parameter string is omitted.

locate

Streams Used: Records are read from the primary input stream and written to the primary output stream. *literal* shorts the input to the output after it has written the argument string to the pipeline.

Record Delay: The first output record is produced before any input is read. Thus, *literal* has the potential to delay one record.

Premature Termination: *literal* terminates when it discovers that its output stream is not connected.

See Also: *strliteral*, *append*, and *var*.

Examples: A literal 3270 data stream is written twice to the console in full screen mode. Hit enter twice to continue. (The left brace represents X'CO'):

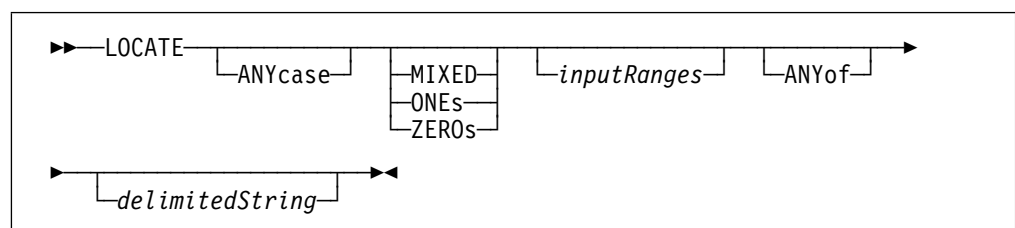
```
PIPE literal {BHit Enter or any PF key | dup | fullscr | hole
```

Notes:

1. Records from a cascade of *literal* stages appear in the reverse order of their appearance in the pipeline specification; see Figure 62 on page 34.
2. Use *var* to write data that contain stage separators, end characters, and other characters that have a special meaning to the pipeline specification parser.
3. *literal* may be used to inject a record in front of the file somewhere downstream in a pipeline, but it can also be a first stage. Note that if you wish to insert a record in front of a file that comes from disk, you must retain the *disk* stage as the first in the pipeline. If not, *disk* appends the single record to the file instead of reading from the file.
4. Be careful when *literal* is used where the contents of a stemmed array are being updated or in similar situations where the output overwrites the original data source. Because *literal* writes the first record before it reads input, this record may be produced before the input has been read; thus, the first record of the updated object may be written before it is read, leading to a “destructive overlap”.
5. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

locate—Select Lines that Contain a String

locate selects records that contain a specified string or that are at least as long as a specified length. It discards records that do not contain the specified string or that are shorter than the specified length.



Type: Selection stage.

Syntax Description:

ANYCASE	Ignore case when comparing.
MIXED	The delimited string is to be used as a mask to test for all ones, mixed, or all zero bits. Only bit positions that correspond to one bits in the mask are tested; bit positions corresponding to zero bits in the mask are ignored. The string must not be null. Records are selected if the delimited string satisfies the condition somewhere within the specified ranges.
ONES	
ZEROS	
	ANYOF cannot be specified with one of these keywords.
ANYOF	The delimited string specifies an enumerated set of characters rather than a string of characters. <i>locate</i> selects records that contain at least one of the enumerated characters within the specified input ranges.

No input range, a single input range, or one to ten input ranges in parentheses can be specified. The default is to search the complete input record.

The characters to search for are specified as a delimited string. A null string is assumed when the delimited string is omitted.

Operation: *locate* copies records in which the specified string occurs within any of the specified input ranges to the primary output stream, or discards them if the primary output stream is not connected. It discards records that do not contain the string within any of the input ranges or that do not include any positions in any of the specified column ranges or copies them to the secondary output stream if it is connected. Thus, it discards null records.

A null string matches any record. In this case, records selected are long enough to include the first position of the input range closest to the beginning of the record. This is used to select records of a given length or longer. Records of a particular length can be selected by a cascade of *locate* and *nlocate*.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *locate* strictly does not delay the record.

Commit Level: *locate* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *locate* terminates when it discovers that no output stream is connected.

Converse Operation: *nlocate*.

See Also: *all* and *find*.

Examples: To discard null records:

```
pipe literal abc | literal | locate | console
▶abc
▶Ready;
```

To select records with 'ab' anywhere in columns 1-4:

lookup

```
pipe literal xaby | literal xxyab | locate 1.4 /ab/ | console
▶xaby
▶Ready;
```

To select records with 'ab' in columns 1-2 or 3-4:

```
pipe literal xaby abcd cdab | split | locate (1-2 3-4) /ab/ | console
▶abcd
▶cdab
▶Ready;
```

To select records that contain X'02' in column 1:

```
pipe < pipparm txtlib | locate 1 x02 | count lines | console
▶2414
▶Ready;
pipe < pipparm txtlib | count lines | console
▶2449
▶Ready;
```

Notes:

1. Use a cascade of *locate* filters or *all* when looking for records containing two or more strings that may occur in any order.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
3. Specifying MIXED and a mask that contains less than two one bits in any one byte will cause all records to be rejected.

lookup—Find Records in a Reference Using a Key Field

lookup processes an input stream of detail records against a reference that contains master records, comparing a key field:

- When a detail record has the same key as a master record, the detail record or the master record (or both) are passed to the primary output stream.
- When a detail record has a key that is not present in any master record, it is passed to the secondary output stream.
- When all detail records have been processed, master records are passed to the tertiary output stream. If COUNT is specified, all master records are written; each one is prefixed by the count of matching detail records. If COUNT is omitted, only those master records for which there was no corresponding detail record are written.

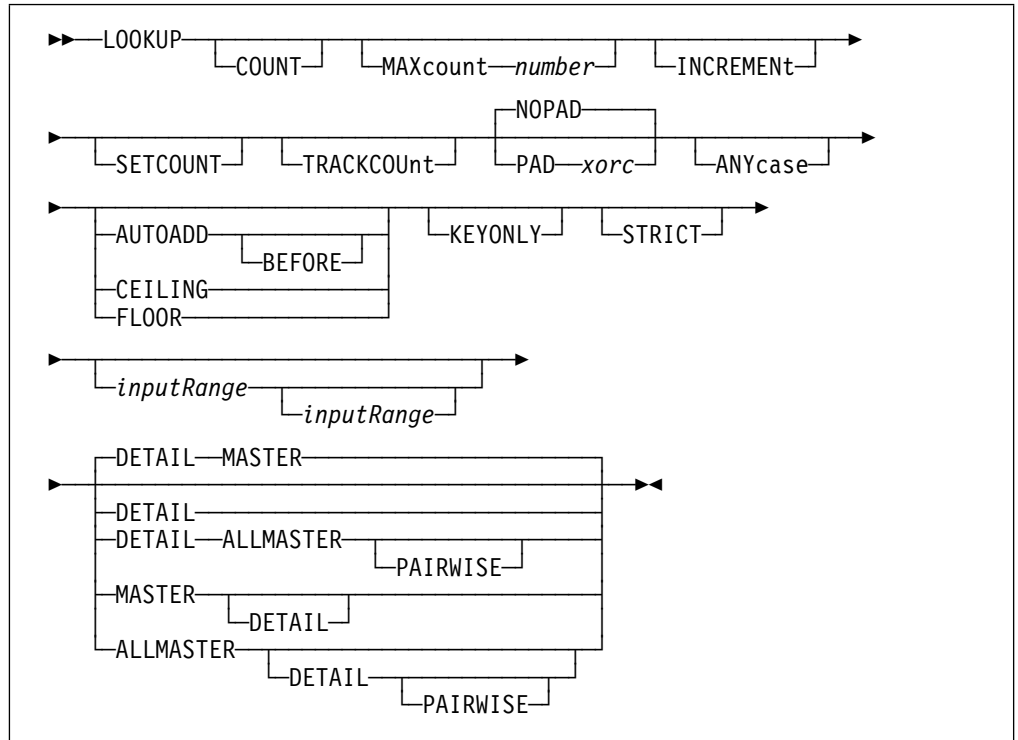
The reference initially contains records from the secondary input stream; this stream is read to end-of-file before processing detail records. When ALLMASTERS is specified, the reference contains all records from the secondary input stream, including those that have duplicate keys; when ALLMASTERS is omitted, *lookup* stores the first record that has a particular key in the reference.

The reference can be updated dynamically in several ways while *lookup* is processing detail records from the primary input stream:

- When AUTOADD is specified, detail records that are not matched are added to the reference automatically.
- Records on the tertiary input stream are added to the reference as they are read.
- Records on the quarternary input stream cause the corresponding reference record(s) to be deleted from the reference.

- Records on the senary input stream replace the corresponding reference record(s).

A count is maintained in each master record irrespective of the COUNT option. By default, one is added for each detail record that matches a particular master record.



Type: Sorter.

Syntax Description: Arguments are optional. The arguments are in three groups:

- Keywords that specify variations on processing.

COUNT A count of matching details is kept with the master record. The count is prefixed to the master record before it is written to the tertiary output stream and the quarternary output stream. When COUNT is omitted, only master records that have a count of zero are written to these two output streams.

MAXCOUNT A master record is deleted after it has been matched by a detail record when its match count is equal to or exceeds the specified number. The number must be positive.

INCREMENT Records on the primary input stream contain the increment in the first ten columns. The number may be negative; it may have leading or trailing blanks. A blank field represents the default increment, one. This number is added to the master's count when the record is matched. The prefix is deleted before the record is matched and written to an output stream; it should be ignored when specifying the range for the key field in the detail record.

SETCOUNT	Records on the secondary input stream and the tertiary input stream contain the initial count in the first ten columns. The number must be zero or positive; it may have leading or trailing blanks. A blank field represents the default starting count, zero. The prefix is deleted before the record is matched and entered into the reference; it should be ignored when specifying the range for the key field in the master record.
TRACKCOUNT	The current count after it has been incremented is prefixed to the master record before it is written to the primary output stream, the tertiary output stream, and the quarternary output stream.
NOPAD	Key fields that are partially present in a record must have the same length to be considered equal; this is the default.
PAD	Specify a pad character that is used to extend the shorter of two key fields.
ANYCASE	Ignore case when comparing fields; the default is to respect case.
AUTOADD	Unmatched details are added to the reference after they have been written to the secondary output stream. If two input ranges are specified, they must be identical.
BEFORE	When AUTOADD BEFORE is specified, the detail record is added to the reference before it is tested, so that it will always be found. Thus, the count will be one when the record is added to the reference; it will be zero when BEFORE is omitted.
: . . .	CEILING
	A detail matches a master record or the master record that has the next higher key. The input record is unmatched only when its key is larger than the highest master key.
: . .	FLOOR
	A detail matches a master record or the master record that has the next lower key. The input record is unmatched only when its key is lower than the lowest master key.
	KEYONLY
	Only the key field is stored in the reference file. Thus, only the key is available to be written to the primary output stream, the tertiary output stream, and the quarternary output stream.
	STRICT
	When records are available simultaneously on more than one input stream, process records from the tertiary input stream before records from the quarternary input stream before records from the primary input stream.

- Input ranges, which specify the location of the key fields in the detail and master records.

The first input range specifies the location of the key in records from the primary input stream (the detail records); the complete input record is the default. The second input range specifies the location of the key in records that are read from other input streams (the master records). If the second range is omitted, the first range is used for the master records as well. When AUTOADD is specified, the second range must be omitted or must represent the same range as the first one.

- Keywords that specify how records are written to the primary output stream when a detail record contains a key that is also in the reference.

DETAIL	Write only detail records to the primary output stream.
DETAIL MASTER	Duplicate master records are discarded. Write the detail record followed by the matching reference to the primary output stream.
DETAIL ALLMASTER	Duplicate master records are kept. Write the detail record followed by all matching masters to the primary output stream.
DETAIL ALLMASTER PAIRWISE	Duplicate master records are kept. For each master record having the selected key, write a copy of the detail record followed by the master record to the primary output stream.
MASTER	Duplicate master records are discarded. Write the matching reference to the primary output stream. The matching detail record is discarded.
MASTER DETAIL	Duplicate master records are discarded. Write the matching reference followed by the detail record to the primary output stream.
ALLMASTER	Duplicate master records are kept. Write all matching master records to the primary output stream. The matching detail record is discarded.
ALLMASTER DETAIL	Duplicate master records are kept. Write all matching master records followed by the detail record to the primary output stream.
ALLMASTER DETAIL PAIRWISE	Duplicate master records are kept. For each master record having the selected key, write the master record followed by a copy of the detail record to the primary output stream.

Operation: The secondary input stream is read and stored as the initial reference before the other streams are read. When ALLMASTER is specified, all master records are stored. When ALLMASTER is omitted, records on the secondary input stream that have duplicate keys are passed to the quinary output stream (if it is defined and connected) or discarded; the first record that has a particular key is retained.

The other input streams are then read as records arrive and processed in this way:

Primary Input Stream: When a record is read on the primary input stream, the contents of the first input range are used as the key. The key field of this detail record is looked up in the reference. When there is no matching master record, the detail record is passed to the secondary output stream (if it is connected). When there is a matching master record, one or more records are written to the primary output stream in the order specified by the keywords DETAIL, MASTER, ALLMASTER, or PAIRWISE. The default is to write the detail record followed by the master record. When MAXCOUNT is specified, an automatic delete is triggered when the count of matches reaches or exceeds the specified number.

Tertiary Input Stream: When a record is read on the tertiary input stream, the contents of the second input range are used as the key. The record is added to the reference if there is not already a record in the reference for the key. The record is also added if ALLMASTERS is specified. Otherwise the record is a duplicate and it is passed to the quinary output stream (if it is defined and connected).

Quarternary Input Stream: When a record is read on the quarternary input stream, the contents of the second input range are used as the key. The corresponding records are deleted from the reference. If there is no matching master record, the input record is passed to the senary output stream (if it is defined and connected). If the quarternary output stream is connected, the corresponding reference record(s) are written to this stream according to the rules stipulated for COUNT; the input record is then discarded. Once the

lookup

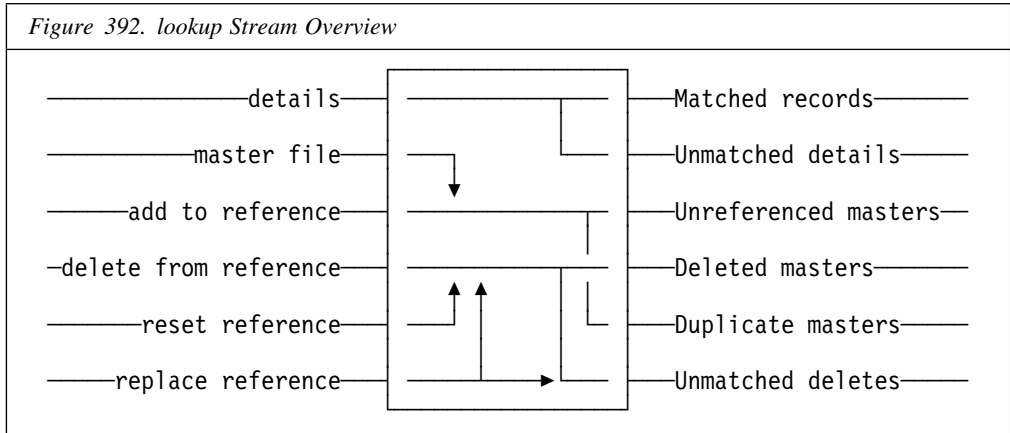
records are deleted from the reference, the count is lost; a subsequent record on the tertiary input stream will start with a reference count of zero.

: *Quinary Input Stream:* When a record is read on the quinary input stream, the master is
 : reset. Before resetting the master, *lookup* writes records to the quarternary output
 : describing the contents of the master file, as it does on the tertiary output prior to termi-
 : nation. All streams remain connected. The record is then discarded.

: *Senary Input Stream:* When a record is read on the senary input stream, it is treated as if
 : the record were passed to the quarternary input stream and then on the tertiary input
 : stream.

At end-of-file on all input streams, all streams other than the tertiary output stream are severed. The contents of the reference (originally from the secondary input stream and tertiary input stream) are then written to the tertiary output stream (if it is connected) in ascending order by their keys. Without the COUNT option, only unreferenced master records are written (those not matched by at least one detail record). When COUNT is specified, all master records are written to the tertiary output stream; they have a 10-byte prefix containing the count of primary input records that matched the key of the master record. Unreferenced records have a count of zero.

: *Streams Used:* Two to six streams can be defined; with AUTOADD, the secondary streams
 : need not be defined. If it is defined, the senary input streams must not be connected.



In Figure 392, the inside of the box shows how records on output streams are derived from input streams, except for the master record being written to the primary output stream; it also shows how end-of-file propagates forward.

All records are read from the secondary input stream before *lookup* reads from other input streams.

When the tertiary output stream is connected, *lookup* severs all other streams at end-of-file on all input streams. It then writes the unreferenced master records to the tertiary output stream (or all master records if COUNT is specified). When both ALLMASTER and COUNT are specified, the count of matching detail records is prefixed to all master records; thus, the sum of the counts will in general be larger than the count of detail records.

lookup propagates end-of-file from the primary input stream to the primary output stream and the secondary output stream; it propagates end-of-file on all first three output streams to the primary input stream; it propagates end-of-file from the tertiary input stream to the

quinary output stream; it propagates end-of-file from the quaternary input stream to the quaternary and senary output streams; it ignores end-of-file on the quinary input stream.

Record Delay: *lookup* features a chaotic delay structure. It does not delay records from the primary input stream (that is, detail records written to the primary output stream or the secondary output stream); nor does it delay records written to the quinary and the senary output stream. Records are written to the tertiary output stream after end-of-file on all input streams; thus, these records are delayed to end-of-file. Records are written to the quaternary output stream before the corresponding input record is consumed from the quaternary input stream.

Commit Level: *lookup* starts on commit level -2. It verifies that the quinary and senary input streams are not connected and then commits to 0.

Premature Termination: *lookup* terminates when it discovers that no output stream is connected.

See Also: *collate*, *merge*, and *sort*.

Examples: The generic EXEC that uses *lookup* with two input streams and three output streams:

```
/* Dictionary lookup */
'PIPE (end ?)',
  '?< detail records',          /* Read details */
  '|l: lookup 1.10',           /* Look up first ten columns */
  '|> matching records a',     /* Details followed by masters */
  '?< master records',        /* Read master file */
  '|l:',                       /* Secondary streams */
  '|> unmatched details a',    /* Details that didn't match */
  '?l:',                       /* Tertiary streams */
  '|> unreferenced masters a' /* Masters that were not referenced */
```

To find all words in the primary input stream that are not in the file WORD LIST:

```
/* CKWB REXX */
'callpipe (end ?)',
  '|*:',                       /* Input stream */
  '|split',                   /* Make words */
  '|l:lookup',               /* Look them up */
  '?< word list',           /* Word list */
  '|split',                 /* One word per line */
  '|l:',                   /* Into master */
  '|*:'                     /* Words not in list to output */
```

Note that the primary output stream from *lookup* is not connected, but that both secondary streams are connected. Also note that there is only one end character in this pipeline specification. The master file is passed into the label reference; the unmatched details come out of the label reference.

To select the first occurrence of each key within the file and not delay the record:

```
/* Now find uniques */
'callpipe (end ?) *: | l:lookup 1.5 autoadd keyonly ? l: | *:'
```

The only connected streams are the primary input stream and the secondary output stream. Thus, the first time a particular key occurs, the detail record will not be matched; it is

lookup

written to the secondary output stream. It is also added to the reference so that subsequent occurrences of that particular key will match and thus, these records will be discarded.

This use of *lookup* has several advantages over *sort* UNIQUE:

- *lookup* does not reorder the records. That is, the output is in the same order as the input (except, of course, that some records are discarded).
- *lookup* does not delay the record.
- *lookup* propagates end-of-file backwards, whereas *sort* cannot produce output until it has read the entire file.
- *lookup* stores only the key field, whereas *sort* must store the entire record. For long records with short keys, this may mean that *lookup* can process larger files than *sort* can.

However, when the entire file is processed (and the key field is not significantly shorter than the record), *lookup* requires as much storage as *sort* UNIQUE and performance will be similar.

To build records for loading several stemmed arrays concurrently in *varset*:

```
callpipe (end ?)
?*:
|l: lookup count trackcount autoadd before keyonly w1 master detail
| spec /=/ 1 11-* n ./ n 1.10 strip n /=/ next read w2-* n
|i: fanin
|*:
?l:
?l:
| spec /=/ 1 11-* n /.0=/ n 1.10 strip n
|i:
```

This subroutine pipeline is provided as a convenience under the name *stembuild*.

In a service machine that maintains privileges for its clients, the immediate commands ADD and DELETE add and delete authorisations dynamically:

```

/* Simplistic server */
'CP SET MSG IUCV' /* Enable commands */
'PIPE (end ? name LOOKUP.STAGE:489)',
  '?starmsg:', /* Requests here */
  '|not chop 8', /* Drop message class */
  '|l:lookup count 1.8 master detail', /* See if allowed */
  '|...', /* Do it! */
  '?< auth file', /* Current authorisations */
  '|l:',
  '|timestamp 16', /* Let's remember when */
  '|>> unauth attempts', /* Log hacking attempts */
  '?imcmd add', /* Immediate commands */
  '|spec w1 1 w2-* 9', /* Build key */
  '|xlate 1.8 upper', /* Uppercase user ID */
  '|l:', /* Add to reference */
  '|not chop 10', /* Delete count */
  '|> auth file a', /* Save updated master */
  '?imcmd delete', /* Immediate command */
  '|spec w1 1.8', /* Just the user id */
  '|xlate', /* Uppercase it */
  '|l:', /* And remove from ref */
  '|insert /Deleted: /', /* Add some text */
  '|console'

```

In this example, the first four input streams to *lookup* are connected. Requests from the users arrive on the primary input stream; the existing authorisations are read into the secondary input stream; new users are authorised by records on the tertiary input stream; and authorisations are dropped by records on the quaternary input stream.

COUNT is specified to have all master records written when the server terminates; but the count is discarded by the *chop* stage that is connected to the secondary output stream. Thus, the current master file can be saved when the *lookup* stage terminates.

The example should not be taken as an example in writing a robust server since it ignores issues such as terminating the server and recovery in the event of a system crash (any added or deleted authorisations would be lost if the virtual machine were reset).

Notes:

1. For compatibility with the past, BOTH specifies the default of writing the detail record followed by the master record to the primary output stream. MATCHING has the same effect as DETAIL; only records from the primary input stream are written to the primary output stream.
2. When the keyword NOPAD is used, key fields must be of the same length to match. Use PAD to specify a character to extend the shorter of two fields when comparing them.
3. Unless ANYCASE is specified, key fields are compared as character data using the IBM System/360 collating sequence.
4. Use *spec* (or a REXX program) for example to put a sort key in front of the record if you wish, for instance, to use a numeric field that is not aligned to the right within a column range. Such a temporary sort key can be removed with *substr* for example after the records are written by *lookup*.
5. Use *xlate* to change the collating sequence of the file.
6. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

lookup

7. *lookup* supports only one key field (unlike *sort*). Use *spec* to gather several key fields into one. When this temporary key is placed in front of the original record, it can be removed from the output of *lookup* easily with *substr* even when the input records vary in length or fields get added at some later time.
8. Counters are “sticky” at zero and the maximum value for a fullword integer. That is, when a counter is decremented below zero, its value is forced to zero; a counter is not incremented beyond 2147483647 ($2^{31}-1$).
9. The five counting options are independent to allow you complete control. In particular, COUNT is not implied by any of the other four.

Thus INCREMENT without COUNT or TRACKCOUNT causes a counter field to be validated as a number and then be deleted from the input detail record. SETCOUNT works similarly for the input master record.

10. When COUNT is omitted, a master record is written to the tertiary output stream or to the quarternary output stream only when its count is zero. When INCREMENT is specified and the increment is zero, the reference count will remain zero and thus the master record will be considered to be unreferenced. Similarly, if SETCOUNT is specified and a master record is added with a nonzero reference count, that record is not written to the tertiary output stream or to the quarternary output stream, even when there are no matching details.
11. When the primary input stream and the secondary input stream are derived from the same source, for example, a selection stage or even another *lookup*, you must buffer the primary input stream to avoid a stall:

```
/* strange lookup */
'PIPE (end ?)'
  '?... ',
  '| x: locate /oscar/ ',
  '| buffer ',
  '| l: lookup ',
  '| ... ',
  '? x: ',
  '| l:
```

Be sure that AUTOADD cannot perform the task.

12. You can replace a record in the reference by passing the replacement on the senary input stream or you can pass the new master record first to the quarternary input stream and then to the tertiary input stream. This destroys the count; to keep the count, you must use the COUNT and SETCOUNT options and preface the first ten characters of the output record to the new master before it is passed to the quarternary input stream (or somehow guess what the count should be and pass that value to the senary input stream).
13. AUTOADD, CEILING, and FLOOR are mutually exclusive.
14. FLOOR and CEILING are useful, for example, to find the control section that contains a particular address to relate trace data to a load map. It might be easiest to convert the keys to binary, but other transforms are possible.
15. Storage for master records that are deleted by passing a record on the quarternary input stream is not reclaimed until *lookup* terminates.

When the master record is replaced by passing a record to the senary input stream, the existing storage is reused if the two records are the same length, rounded to the next multiple of four. Any additional master record for the particular key are not reclaimed.

maclib—Generate a Macro Library from Stacked Members in a COPY File

maclib generates the contents of a CMS macro library from stacked members with delimiter records between them, as the CMS command MACLIB does for an input file that has file type COPY.



Type: Arcane gateway.

Placement: *maclib* must not be a first stage.

Syntax Description: A word is optional. It specifies the delimiter word that separates members in the input stream. '*COPY' is the default.

Operation: *maclib* first writes an 80-byte placeholder record indicating a null library to the primary output stream. It then writes the members of the MACLIB with a delimiter record (X'61FFFF61') after each member. 80-byte directory records (having 16-byte entries) are written to the secondary output stream (if it is connected) for each five members.

At end-of-file on input, the final directory record is written to the secondary output stream and the correct record 1 for the library is written to the tertiary output stream (if it is connected).

Input Record Format: The format is similar to the format of a file with file type COPY, as used by the CMS command MACLIB. The input stream has one or more members, each preceded by a delimiter record with the delimiter word in column 1. The member name is the second word of a line beginning with the delimiter word; the remainder of the line is ignored.

Streams Used: One to three streams may be defined. Records are read from the primary input stream; no other input stream may be connected. *maclib* writes output to all connected output streams. It severs the primary output stream at end-of-file on input before it writes to the secondary output stream. It severs the secondary output stream before it writes to the tertiary output stream.

Record Delay: The first output record is produced before any input is read. Thus, *maclib* has the potential to delay one record. Records are written to the primary output stream before they are consumed from the primary input stream.

Commit Level: *maclib* starts on commit level -2. *maclib* verifies that the primary input stream is the only connected input stream and then commits to 0.

Examples: Refer to “Generating a CMS Macro Library” on page 80.

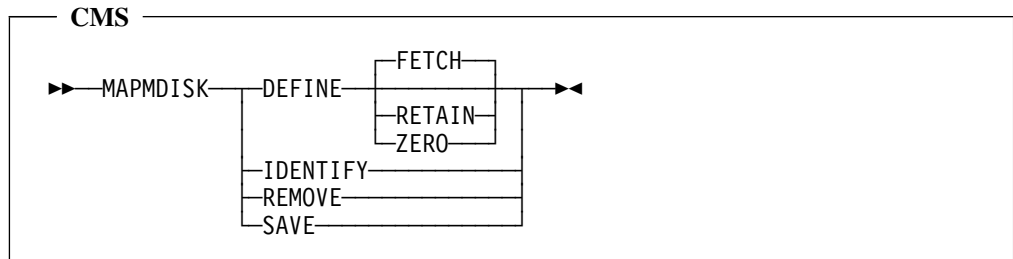
Notes:

1. To create a CMS macro library, the secondary output stream (which contains the directory) should be buffered and appended to the contents of the primary output stream; this aggregate stream should be connected to the primary input stream of the > stage writing the library; *maclib*'s tertiary output stream should be connected to the secondary input stream of >.

2. *maclib* generates the data records to be put into a library. It accesses no host interface; in particular, it does not write the library to disk.

mapmdisk—Map Minidisks Into Data spaces

mapmdisk interfaces to the CP macro MAPMDISK to manage minidisks mapped into address spaces available to your virtual machine.



Type: Host interface.

Syntax Description:

DEFINE	Map blocks from the minidisk pool into an address space. The pool must have been established previously by <i>mapmdisk</i> IDENTIFY.
FETCH	The initial contents of the mapped pages are read from the minidisks.
RETAIN	The initial contents of the mapped pages are left intact from the data space.
ZERO	The initial contents of the mapped pages are set to zero.
IDENTIFY	Define the virtual machine's minidisk pool. <i>mapmdisk</i> IDENTIFY reads its entire input to define the pool.
REMOVE	Unmap minidisk blocks from an address space.
SAVE	Write modified pages in the data space to backing store. This is the only way to ensure a consistent minidisk image.

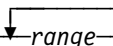
Input Record Format:

CREATE ▶▶—hexString—number—number—number—▶▶

Each record specifies the mapping of a single data space that must have been created by your virtual machine.

1. The ASIT for the data space that blocks are mapped into.
2. The number of the first page to map in decimal (zero or positive). The number is multiplied by 4096 before it is stored in the parameter list.
3. The count of pages to map in decimal (positive).
4. The number of the first pool block to map in decimal (zero or positive).


```

:
: IDENTIFY      ►►—devaddr—number—number—◄◄
:
: Each record defines one minidisk extent in the pool. The pool block
: numbers are assigned sequentially from zero as the input is read.
:
: 1. The device number.
:
: 2. The block offset (decimal, zero or positive), which can be obtained
: by diskid.
:
: 3. The count of blocks (positive decimal), which can be obtained by
: state at the time the minidisk is reserved.
:
: REMOVE      ►►—hexString—number—number—◄◄
:
: Each record specifies a range of pages to be unmapped.
:
: 1. The ASIT for the data space from which blocks are unmapped.
:
: 2. The number of the first page to unmap in decimal (zero or positive).
: The number is multiplied by 4096 before it is stored in the param-
: eter list.
:
: 3. The count of pages to unmap in decimal (positive).
:
: SAVE
:
: ►►—hexString——◄◄
:
: Each record specifies a range of blocks to be saved, if they have been
: changed. The first word is the ASIT of the address space to save. Then
: follows up to 509 ranges of pages to be saved. The first number in
: each range is multiplied by 4096 before it is stored in the parameter list.
:
:
: Output Record Format:
:
: CREATE      The input record is passed.
:
: IDENTIFY    A null record to indicate that a pool has been defined.
:
: REMOVE      The input record is passed.
:
: SAVE        Ten bytes:
:
: • The eight byte error buffer, which is valid when the leftmost bit of
: the completion status (the last byte of the record) is zero.
:
: • Binary interrupt subcode from storage location 132 (decimal, one
: byte), X'01'.
:
: • Binary completion status from storage location 133 (decimal, one
: byte). The contents of the first eight bytes of the record are valid
: when the leftmost bit is zero.
:
:
: Streams Used: Records are read from the primary input stream and written to the primary
: output stream. Null and blank input records are discarded.
:
:
: Record Delay: mapmdisk does not delay the record.

```

mctoasa

: **Commit Level:** *mapmdisk* starts on commit level 0 or -2. In general, it starts on commit
: level 0, but *mapmdisk* SAVE starts on commit level -2, establishes an external interrupt
: handler, and then commits to level 0.

: **Premature Termination:** *mapmdisk* terminates when it discovers that its output stream is
: not connected. *mapmdisk* SAVE also stops if the immediate command PIPMOD STOP is
: issued or a record is passed to *pipestop*.

: **See Also:** *adrspac*, *alserv*, and *diskid*.

: **Examples:** See Chapter 18, “Using VM Data Spaces with *CMS Pipelines*” on page 210.

: **Notes:**

1. The virtual machine must be in XC mode (this excludes z/CMS).
2. All minidisk mappings are destroyed by an IPL of the virtual machine.

: **Publications:** z/VM: CP Programming Services.

mctoasa—Convert CCW Operation Codes to ASA Carriage Control

mctoasa converts the first byte of each record from a machine carriage control character to an ASA control character. If possible, the ASA control character is moved to the following record to turn a delayed carriage movement into an immediate one. If the first input record has a valid ASA carriage control character, the input is passed unmodified to the output and each record is verified to have a valid ASA carriage control character.

▶▶—MCTOASA—▶▶

Type: Filter.

Input Record Format: The first column of the record is a machine carriage control character:

xxxx x001	Write the data part of the record and then perform the carriage operation specified by the five leftmost bits.
xxxx x011	Perform the carriage operation defined by the five leftmost bits immediately (the data part of the record is ignored).
000n n0x1	Space the number of lines (0 through 3) specified by bits 3 and 4.
1nnn n0x1	Skip to the channel specified by bits 1 through 4. The number must be in the range 1 to 12 inclusive.

Other bit combinations are not valid. In particular, bit 5 (X'04') must be zero.

Output Record Format: The first column of the record is an ASA carriage control character:

+	Overprint the previous record.
(blank)	Print on the next line.
0	Skip one line and print a line.
- (hyphen)	Skip two lines and print a line.
1-9	Skip to the specified channel and print a line.
A-C	Skip to channel 10 through 12 and print a line.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: When the file has machine carriage control characters, the carriage control is delayed to the following record; the data part of the record is not delayed.

Premature Termination: *mctoasa* terminates when it discovers that its output stream is not connected.

Converse Operation: *asatmc*.

Examples: To discard the two heading lines from an assembler listing file:

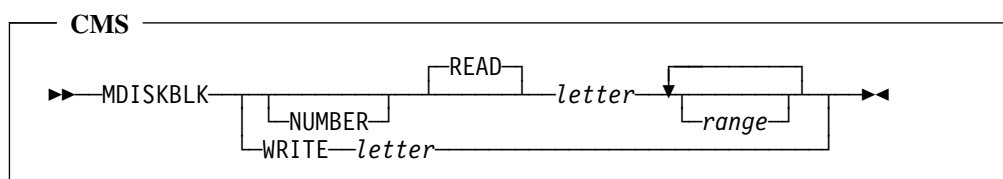
```
...| mctoasa | outside /1/ 2 |...
```

The System Assembler generates the listing file with ASA carriage control; assembler H uses machine carriage control. *mctoasa* ensures that the listing file has ASA carriage control characters in either case.

mdiskblk—Read or Write Minidisk Blocks

mdiskblk reads and writes physical data blocks directly from a minidisk, bypassing the CMS file system. Be sure you know what you are doing when you use *mdiskblk*!

Warning: Improper use of *mdiskblk* WRITE may result in an unreadable minidisk.



Type: Arcane device driver.

Syntax Description: When READ is specified or defaulted, a letter is required to specify the mode of the minidisk to read blocks from; further arguments are ranges of blocks to be read.

When WRITE is specified, only a mode letter is allowed.

Operation: When *mdiskblk* is reading blocks from a minidisk, the blocks specified in the argument string (if any) are read into the pipeline. The blocks specified in input records are then read. If NUMBER is specified, each output record is prefixed with a 10-byte field containing the block number of the record.

Input Record Format: When *mdiskblk* is reading blocks from a minidisk, input records must contain blank-delimited ranges that specify the blocks to read from the minidisk. A range that ends with an asterisk (for example, 1-*) extends to the end of the minidisk (or to wherever it has been recomputed with FORMAT RECOMP); a range that specifies a block number beyond the end of the minidisk attracts an error message and causes *mdiskblk* to terminate.

When *mdiskblk* is writing blocks to the minidisk, input records must contain the blocks to write prefixed by a 10-byte record number (in printable decimal). The input record length must be ten more than the disk block size.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *mdiskblk* writes all output for an input record before consuming the input record.

Premature Termination: *mdiskblk* terminates when it discovers that its output stream is not connected.

Examples: To read the label record of the “S” minidisk on a count key data device:

```
pipe mdiskblk s 3 | spec 5.6 1 | console
▶MNT190
▶Ready;
```

To read the top pointer block from a file (or the only data block, when the file contains only one block):

```
/* Read top pointer */
'PIPE',
  ' literal PROFILE EXEC A ',
  '|state noformat |',
  '|spec 41.4 c2d 1|',
  '|mdiskblk a|',
  '|> prof pointer a|'
```

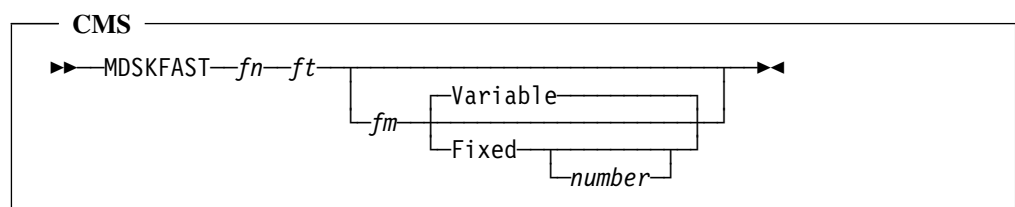
This example fails if the file consists entirely of binary zeros; CMS may elect not to write any disk blocks for such a file.

Notes:

1. The minidisk must be accessed even though the CMS file system is bypassed.
2. Specify the actual ending block number to read a complete minidisk, including the part of the disk that is reserved for a nucleus.

***m*diskfast—Read, Create, or Append to a CMS File on a Mode**

*m*diskfast connects the pipeline to a CMS file on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter. When it is first in a pipeline, *m*diskfast reads a file from disk; it treats a file that does not exist as one with no records (a null file). When it is not first in a pipeline, *m*diskfast appends records to an existing file; a file is created if one does not exist.



Type: Device driver.

Warning: *mdufast* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *mdufast* is unintentionally run other than as a first stage. To use *mdufast* to read data into the pipeline at a position that is not a first stage, specify *mdufast* as the argument of an *append* or *preface* control. For example, `|append mdufast ...|` appends the data produced by *mdufast* to the data on the primary input stream.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be read or appended to. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried. No further arguments may be specified when *mdufast* is first in a pipeline.

When *mdufast* is not first in a pipeline, the file is created as A1 if no file mode (or an asterisk) is specified and no file is found with the name and type given. The record format and (for fixed format files) the record length are optional arguments. The default is the characteristics of an existing file when appending, VARIABLE when a file is being created. When the file exists, the specified record format must match the characteristics of the file.

Operation: When *mdufast* is first in a pipeline, reading starts at the beginning of the file. When *mdufast* is not first in a pipeline, *mdufast* appends records from the primary input stream to an existing file. The file is closed before *mdufast* terminates.

Streams Used: When *mdufast* is first in a pipeline, it writes records to the primary output stream.

When *mdufast* is not first in a pipeline, it first appends to or creates the file from records on the primary input stream that are not null; all input records are also copied to the primary output stream. The primary output stream is severed at end-of-file on the primary input stream. The first records of the file are then overwritten with any records from the secondary input stream that are not null. All records from the secondary input stream are copied to the secondary output stream after they are written to the file.

Warning: When the secondary input stream is connected, records read from it must have the same length as the records they replace in the file, but this is not enforced by CMS for variable record format files; CMS truncates a variable record format file without indication of error if a record is replaced with one of different length, be that shorter or longer.

Record Delay: *mdufast* strictly does not delay the record.

Premature Termination: When it is first in a pipeline, *mdufast* terminates when it discovers that its output stream is not connected.

See Also: `>`, `>>`, `<`, *diskslow*, *diskback*, *diskrandom*, *diskupdate*, *members*, and *pdsdirect*.

Examples: To count the number of words in a file which may or may not exist:

```
pipe mdufast input file | count words | console
```

Notes:

1. Use *diskslow* if *mdufast* fails to operate.
2. Use *diskslow* to begin to read or write from a particular record; use *diskrandom* to read records that are not sequential; use *diskupdate* to replace records in random order.

3. Null input records are copied to the output (if connected), but not to the file; CMS files cannot contain null records.
4. *m*dskfast can read or append to a file with a name in mixed case (if you enter the exact file name and file type), but it creates only files with file names and file types in upper case. Use *command* RENAME to change a file's name or type to mixed case.
5. When it is first in a pipeline, *m*dskfast may obtain several records from CMS at a time. When it is not first in a pipeline and it is processing records from the primary input stream, *m*dskfast may deliver several records at a time to CMS to improve performance. The file may not be in its eventual format while it is being created; it should not be accessed (by any means) before *m*dskfast terminates. It is unspecified how many records *m*dskfast buffers, as well as the conditions under which it does so.
6. Connect the secondary input stream when creating CMS libraries or packed files where the first record has a pointer to the directory or contains the unpacked record length of a packed file. The stage that generates the file (for instance, *maclib*) can write a placeholder first record on the primary output stream initially; it then writes the real first record to a stream connected to the secondary input stream of *m*dskfast when the complete file has been processed and the location and size of the directory are known.
7. The fast interface to the file system is bypassed if the bit X'10' is on in offset X'3D' of the FST that is exposed by the FSSTATE macro. Products that compress files on the fly or in other ways intercept the file system macros should turn on this bit to ensure that *CMS Pipelines* uses documented interfaces only.

Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, *m*dskfast is transparent to return codes from CMS. Refer to the return codes for the FSREAD macro and the FSWRITE macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 1 You do not have write authority to the file.
- 13 The disk is full.
- 16 Conflict when writing a buffer; this indicates that a file with the same name has been created by another stage.
- 20 The file name or file type contains an invalid character.
- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

*m*dskback—Read a CMS File from a Mode Backwards

*m*dskback reads the last record, then the second last record, and so on from a CMS file on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter.



Type: Device driver.

Placement: *m*dskback must be a first stage.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be read. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried.

Operation: The file is closed before *mdskback* terminates.

Premature Termination: *mdskback* terminates when it discovers that its output stream is not connected.

See Also: *<*, *disk*, *diskslow*, *members*, and *pdsdirect*.

Examples: To read the last message from a notebook file and append it to the file being edited:

```
! /* GETLAST XEDIT */
! arg fn .
! 'bottom'
! 'pipe mskback' word(fn 'all', 1) 'notebook a',
!   | strtolabel /=====/',
!   | instore reverse',
!   | outstore',
!   | xedit'
```

Notes:

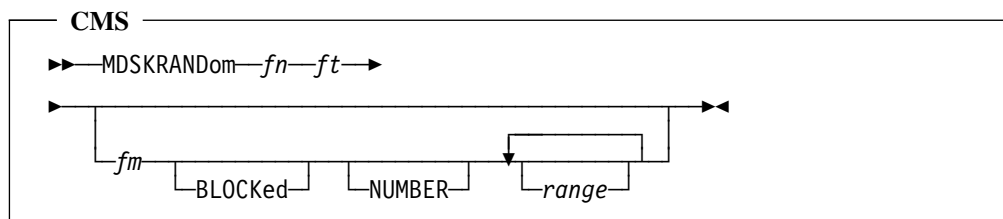
1. For short files it can be more efficient to read the file with *<* and use *instore REVERSE* followed by *outstore* to reverse the order of the records in a file.
2. *mdskback* may obtain several records from CMS at a time. It is unspecified how many records *mdskback* buffers, as well as the conditions under which it does so.

Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, *mdskback* is transparent to return codes from CMS. Refer to the return codes for the FSREAD macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 20 The file name or file type contains an invalid character.
- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

mdskrandom—Random Access a CMS File on a Mode

mdskrandom reads records in a specified order from a CMS file on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter.



Type: Device driver.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be read. A file mode or an asterisk is optional; the default is to search all modes. Use an asterisk as a placeholder for the file mode when you wish to specify further arguments and search all accessed modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried.

BLOCKED	Write a range of records from the file as a single output record; the file must have fixed record format.
NUMBER	Prefix the record number to the output record. The field is ten characters wide; it contains the number with leading zeros suppressed.
<i>range</i>	Further arguments are ranges of records to be read. Use an asterisk as the end of a range to read to the end of the file.

Operation: The records whose numbers are specified in the argument are read into the pipeline. Lines are then read from the input stream (if it is connected). Input records contain blank-delimited words that specify ranges of records to read from the file. Output records are written in the order specified in the argument string and input records. The file is closed before *mdskrandom* terminates.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *mdskrandom* does not delay the last record written for an input record. An input record that contains a single number is not delayed. Nor is an input record that contains a single range, when BLOCKED is specified.

Premature Termination: *mdskrandom* terminates when it discovers that its output stream is not connected.

See Also: `>`, `>>`, `<`, *disk*, *diskback*, *diskslow*, *members*, and *pdsdirect*.

Examples: Both of these commands read records 7, 8, 3, and 1 from a file and write them to the pipeline in that order:

```
pipe mdskrand profile exec * 7.2 3 1 |...
pipe literal 3 1 | mdskrand profile exec * 7.2 |...
```

Notes:

1. RECNO is a synonym for NUMBER.
2. *mdskrandom* performs at least one read operation for the records in the arguments, if specified, and one read operation for each input record. When BLOCKED is specified, all records in a range are read in a single operation. It is unspecified how many additional read operations it performs for records specified in the arguments or a particular input record. This may be significant when the file is updated with *diskupdate*. Ensure that no stage delays the record between stages reading and writing a file being updated.

Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, *mdskrandom* is transparent to return codes from CMS. Refer to the return codes for the FSREAD macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 20 The file name or file type contains an invalid character.

- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

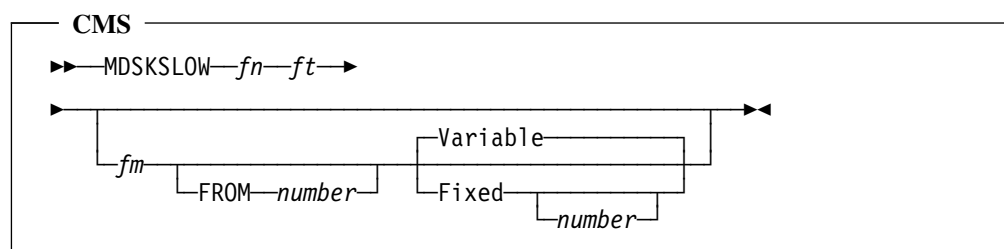
mdskslow—Read, Append to, or Create a CMS File on a Mode

mdskslow connects the pipeline to a CMS file on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter. Unlike *mdskfast*, *mdskslow* does not try to read or write several records at a time.

When it is first in a pipeline, *mdskslow* reads a file from disk; it treats a file that does not exist as one with no records (a null file). When it is not first in a pipeline, *mdskslow* appends records to an existing file; a file is created if one does not exist.

Use *mdskslow* rather than *mdskfast*, <, >, or >>:

- If *disk* (or one of its other entry points) fails to operate.
- If a file is to be written unblocked. This may help to identify which record causes an error (for example, a program check) in a previous or subsequent stage.
- When writing a file from several stages concurrently. (It may be a better idea, however, to use *faninany* to gather the streams and write with a single stage.)
- To begin reading or writing from a particular record number.



Type: Device driver.

Warning: *mdskslow* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *mdskslow* is unintentionally run other than as a first stage. To use *mdskslow* to read data into the pipeline at a position that is not a first stage, specify *mdskslow* as the argument of an *append* or *preface* control. For example, `|append mdskslow ...|` appends the data produced by *mdskslow* to the data on the primary input stream.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be read or appended to. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried. The keyword FROM is optional after the file mode; the following word specifies the number of the first record to read or write; the defaults are to read from the beginning of the file and to append after the last record of the file. No further arguments may be specified when *mdskslow* is first in a pipeline.

When *mdskslow* is not first in a pipeline, the file is created as A1 if no file mode (or an asterisk) is specified and no file is found with the name and type given. The record format and (for fixed format files) the record length are optional arguments. The default is the characteristics of an existing file when appending, VARIABLE when a file is being created. When the file exists, the specified record format must match the characteristics of the file.

Operation: *mdskslow* is similar to *disk*, but it uses the FSREAD and FSWRITE interface to the file system to read and write records, issuing a call for each record. The file is closed before *mdskslow* terminates.

Streams Used: When *mdskslow* is first in a pipeline, it writes records to the primary output stream.

When *mdskslow* is not first in a pipeline, it first appends to or creates the file from records on the primary input stream that are not null; all input records are also copied to the primary output stream. The primary output stream is severed at end-of-file on the primary input stream. The first records of the file are then overwritten with any records from the secondary input stream that are not null. All records from the secondary input stream are copied to the secondary output stream after they are written to the file.

Warning: When the secondary input stream is connected, records read from it must have the same length as the records they replace in the file, but this is not enforced by CMS for variable record format files; CMS truncates a variable record format file without indication of error if a record is replaced with one of different length, be that shorter or longer.

Record Delay: *mdskslow* strictly does not delay the record.

Premature Termination: When it is first in a pipeline, *mdskslow* terminates when it discovers that its output stream is not connected.

See Also: `>`, `>>`, `<`, *disk*, *diskback*, *diskrandom*, *diskupdate*, *members*, and *pdsdirect*.

Examples: To replace parts of a file starting with record 713:

```
pipe < replace part | diskslow old file * from 713
```

Notes:

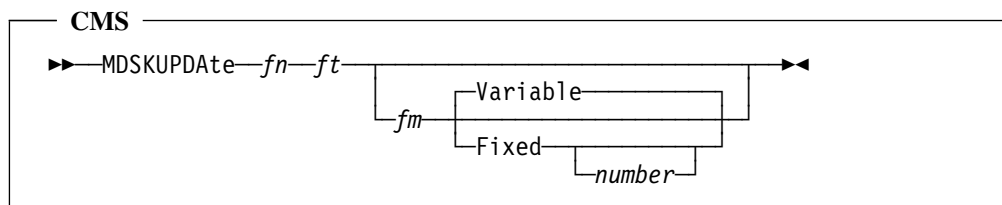
1. Null input records are copied to the output (if connected), but not to the file; CMS files cannot contain null records.
2. *mdskslow* does not buffer or block reads or writes to CMS files.
3. Connect the secondary input stream when creating CMS libraries or packed files where the first record has a pointer to the directory or contains the unpacked record length of a packed file. The stage that generates the file (for instance, *maclib*) can write a placeholder first record on the primary output stream initially; it then writes the real first record to a stream connected to the secondary input stream of *mdskslow* when the complete file has been processed and the location and size of the directory are known.

Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, *mdskslow* is transparent to return codes from CMS. Refer to the return codes for the FSREAD macro and the FSWRITE macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 1 You do not have write authority to the file.
- 13 The disk is full.
- 16 Conflict when writing a buffer; this indicates that a file with the same name has been created by another stage.
- 20 The file name or file type contains an invalid character.
- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

mdskupdate—Replace Records in a File on a Mode

mdskupdate replaces records in or appends records to a CMS file on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter.



Type: Device driver.

Placement: *mdskupdate* must not be a first stage.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be updated. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried. The record format is optional after the file mode; a record length is optional for fixed record format files.

Operation: Columns 11 through the end of the input record replace the contents of the record in the file. The file is closed before *mdskupdate* terminates.

Input Record Format: The first 10 columns of an input record contain the number of the record to replace in the file (the first record has number 1). The number does not need to be aligned in the field. It is an error if an input record is shorter than 11 bytes.

The valid values for the record number depends on the record format of the file:

- Fixed For fixed record format files, any number can be specified for the record number (CMS creates a sparse file if required). An input record can contain any number of consecutive logical records as a block. The block has a single 10-byte prefix containing the record number of the first logical record in the block.
- Variable When the file has variable record format, the record number must be at most one larger than the number of records in the file at the time the record is written to it. The data part of input records must have the same length as the records they replace in the file, but this is not enforced by CMS for variable record format files; CMS truncates a variable record format file without indication of error if a record is replaced with one of different length, be that shorter or longer.

Streams Used: *mdskupdate* copies the input record (including the record number) to the output after the file is updated with the record. must have the same length as the records they replace in the file, but this is not enforced by CMS for variable record format files; CMS truncates a variable record format file without indication of error if a record is replaced with one of different length, be that shorter or longer.

Record Delay: *mdskupdate* strictly does not delay the record.

See Also: `>`, `>>`, *disk*, and *diskslow*.

Examples: To replace records in a file with a particular key:

```
/* Update file */
'PIPE',
  ' diskslow input file',
  '| spec number 1 1-* next',
  '| locate 11.3 /abc/',
  '| spec 1-* 1 /def/ 11',
  '| mdsupdate input file'
```

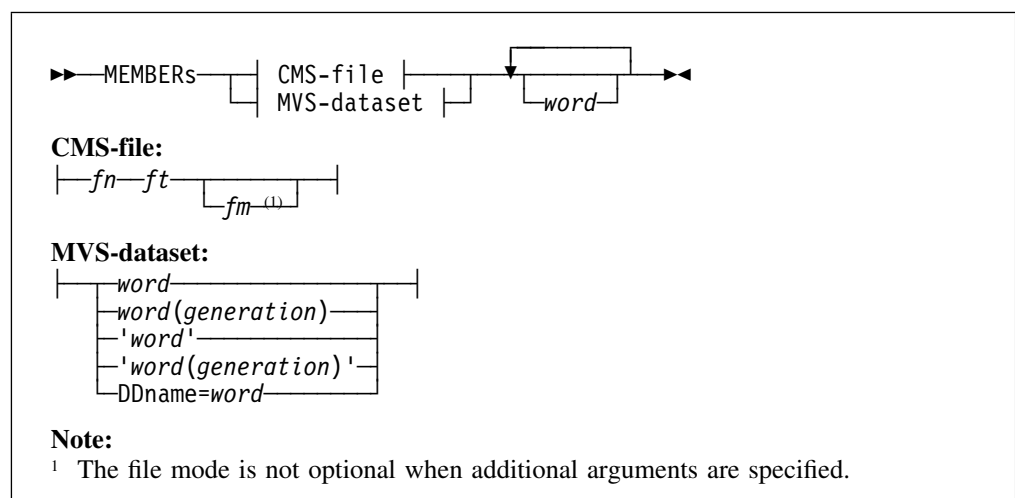
Return Codes: In addition to the return codes associated with *CMS Pipelines* error messages, *mdskupdate* is transparent to return codes from CMS. Refer to the return codes for the FSWRITE macro in *z/VM CMS Macros and Functions Reference*, SC24-6262, for a complete list of return codes. You are most likely to encounter these:

- 1 You do not have write authority to the file.
- 13 The disk is full.
- 16 Conflict when writing a buffer; this indicates that a file with the same name has been created by another stage.
- 20 The file name or file type contains an invalid character.
- 24 The file mode is not valid.
- 25 Insufficient storage for CMS to allocate buffers.

members—Extract Members from a Partitioned Data Set

members reads members from a library into the pipeline. The member names may be specified as arguments, or they may be provided in input records, or both.

On CMS, *members* supports only a MACLIB, TXTLIB, or a similar file that has fixed record format and record length 80. The library must be on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter. The file must exist.



Type: Device driver.

Syntax Description:

CMS Specify as blank-delimited words the file name and the file type of the file to be read. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name

and the file type are translated to upper case and the search is retried. The file must be fixed record format and record length 80.

MVS Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the `DSNAME` without quotes to have the prefix, if any, applied. Append parentheses containing a signed number to specify a relative generation of a data set that is a member of a generation data group.

To read members of an already allocated data set, specify the keyword `DDNAME=` followed by the `DDNAME` already allocated. The minimum abbreviation is `DD=`.

A blank-delimited list of member names is optional after the data set identifier.

Operation: *members* writes the contents of the specified members to the primary output stream in the order specified. The members are specified by the names in the argument string, if any, followed by the input records.

Each member is looked up in the library directory. If the member does not exist as written, the search is retried with the member name translated to upper case.

A null record is written after each member; on CMS, trailing LDT and end of member records are discarded.

In a CMS `TXTLIB`, *members* finds only the “main” name of a member (the first CSECT); additional entry points are not found.

Diagnostic messages are issued for members that are not present in the library; the argument and all input records are processed before returning with return code 150 when one or more members are not found.

The file is closed before *members* terminates.

Input Record Format: Blank-delimited lists of members to read from the library.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *members* writes all output for an input record before consuming the input record.

Commit Level: *members* starts on commit level -2. *members* starts on commit level -1 on CMS. It reads the directory of the library and then commits to level 0. *members* starts on commit level -2000000000 on z/OS. It opens the DCB and then commits to level 0.

Premature Termination: *members* terminates when it discovers that its output stream is not connected.

See Also: *listispf* and *listpds*.

Examples: To extract a member of the system macro library and remove comment lines:

merge

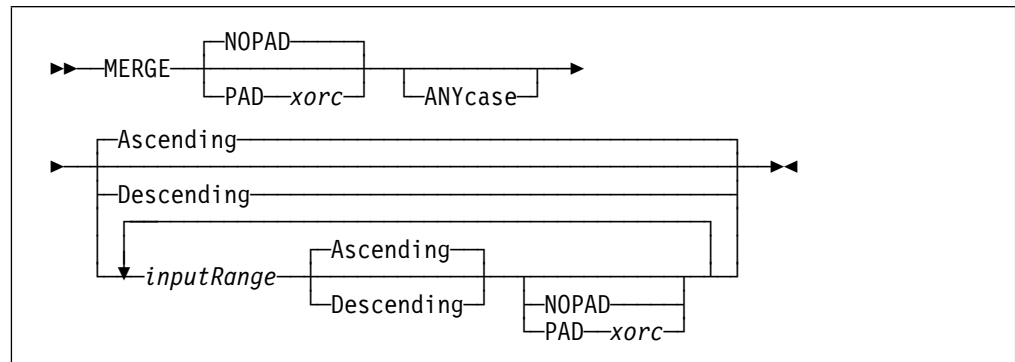
```
!           pipe members dmsom maclib * diag | nfind .*| chop 72 | console
!           ▶          MACRO
!           ▶&LBL     DIAG  &R1,&R2,&CODE
!           ▶&LBL     DC    0H'0',X'83',AL.4(&R1,&R2),Y(&CODE)
!           ▶          MEND
!           ▶
!           ▶
!           ▶Ready;
```

Notes:

1. On CMS, *xtract* (with synonym *extract*) performs the same operation as *members* on a TXTLIB; it is retained for compatibility with the past.
2. On CMS a library can contain members that have the same name. When the library has more than one member by a particular name, it is unspecified which one *members* reads.
3. On z/OS, *members* is a synonym for *readpds*.

merge—Merge Streams

merge merges multiple input streams into a single output stream, interleaving the records according to the contents of their key fields. The input streams should already be sorted in the specified order to produce a sorted output stream; this is not verified.



Type: Sorter.

Syntax Description: Write the keywords PAD or NOPAD in front of the sort fields to specify the default for all fields; the default is NOPAD. The keyword NOPAD specifies that key fields that are partially present must have the same length to be considered equal; this is the default. The keyword PAD specifies a pad character that is used to extend the shorter of two key fields.

The keyword ANYCASE specifies that case is to be ignored when comparing fields; the default is to respect case. Up to 10 sort ranges can be specified. The default is to merge ascending on the complete record. The ordering can be specified for each field; it is ascending by default. Specify padding after the ordering to treat a field differently than other fields.

Operation: Records with identical keys on two or more streams are written with the record from the lowest numbered stream first. This ensures that a sort/merge can be made stable so that multiple sorts of a file give the same result.

Streams Used: Up to 10 input streams (numbered 0 through 9) are supported. Output is written only to the primary output stream.

Record Delay: *merge* consumes an input record after it has been copied to the output. In this sense it does not delay the record, but it clearly allows a record from one input stream to overtake the record on another one.

Commit Level: *merge* starts on commit level -2. It verifies that the only connected output stream is the primary one and then commits to 0.

Premature Termination: *merge* terminates when it discovers that its primary output stream is not connected.

See Also: *sort*.

Examples: To merge two files that are already sorted:

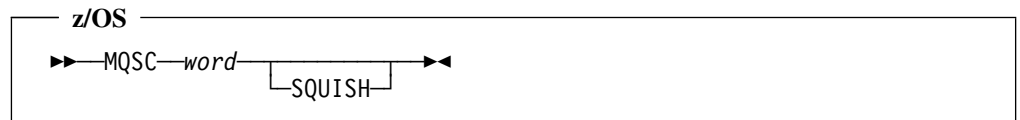
```
pipe (end ?) < first file | m: merge | > big file a ? < second file | m:
```

Notes:

1. Large files can be sorted in parts to disk work files and subsequently merged with *merge*.
2. In early versions of *CMS Pipelines merge* was used to insert a marker in a sorted stream of records so *tolabel* and *flabel* could be used to emulate “less than” or “greater than” comparison to select records. In most cases *pick* can simplify such a pipeline without the need to sort the records.
3. Unless ANYCASE is specified, key fields are compared as character data using the IBM System/360 collating sequence.
4. Use *spec* (or a REXX program) for example to put a sort key in front of the record if you wish, for instance, to use a numeric field that is not aligned to the right within a column range. Such a temporary sort key can be removed with *substr* for example after the records are written by *merge*.
5. Use *xlate* to change the collating sequence of the file.
6. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

mqsc—Issue Commands to a WebSphere MQ Queue Manager

mqsc sends requests to a queue manager’s command queue and writes the response to the primary output stream.



Type: Device driver.

Syntax Description:

nfind

word Specify the subsystem ID of the queue manager to access (four characters). Case is respected in the first operand; most z/OS subsystems have upper case IDs.

SQUISH Replace multiple blanks in the response with a single one.

Input Record Format: The input should contain MQ Command Script commands, as defined in the MQ Commands Reference manual.

Output Record Format: The output contains the response from the command processor. In general, there will be one line for each object processed.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *mjsc* does not delay the record.

Commit Level: *mjsc* starts on commit level -20. It connects to the queue manager if *TSO Pipelines* is not already connected to a queue manager. It then opens the queues it needs and then commits to level 0.

Premature Termination: *mjsc* terminates when it discovers that its output stream is not connected.

Examples:

```
pipe literal display qmgr|mjsc MQA1|cons
▶CSQM409I +MQA1 QMNAME(MQA1 )
▶CSQ9022I +MQA1 CSQMDRTS ' DISPLAY QMGR' NORMAL COMPLETION
▶READY
```

Notes:

1. Multiple instances of *mjsc* may run concurrently as long as they all refer to the same queue manager.
2. Some commands, such as PING CHANNEL are asynchronous; their response indicates that the operation has been started, but the result of the operation is unknown.

nfind—Select Lines by XEDIT NFind Logic

nfind selects records that do not begin with the specified string. It discards records that begin with the specified string. XEDIT rules for NFIND apply.



Type: Selection stage.

Syntax Description: A string is optional. The string starts after exactly one blank character. Leading and trailing blanks are significant.

Operation: Input records are matched the same way XEDIT matches text in an NFIND command (tabs 1, image off, case mixed respect):

- A null string matches any record.

- Blank characters in the string represent positions that must be present in the input record, but can have any value.
- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

nfind copies records that do not match to the primary output stream, or discards them if the primary output stream is not connected. It discards records that match or copies them to the secondary output stream if it is connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *nfind* strictly does not delay the record.

Commit Level: *nfind* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *nfind* terminates when it discovers that no output stream is connected.

Converse Operation: *find*.

See Also: *nlocate* and *strnfind*.

Examples: To discard lines with 'a' in column 1 and 'c' in column 3:

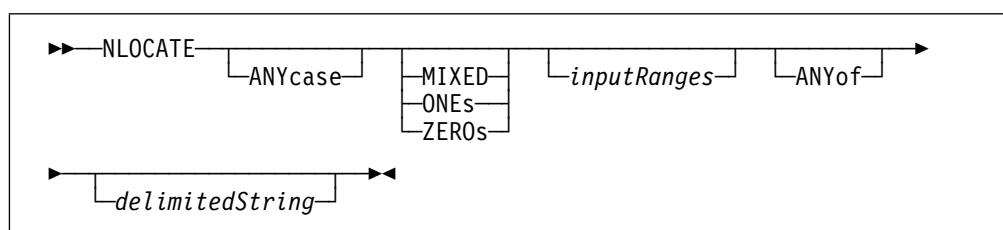
```
pipe literal abc axc axy xyc | split | nfind a c | console
▶axy
▶xyc
▶Ready;
```

Notes:

1. *notfind* is a synonym for *nfind*.
2. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

nlocate—Select Lines that Do Not Contain a String

nlocate discards records that contain a specified string or are at least as long as a specified length. It selects records that do not contain the specified string or that are shorter than the specified length.



Type: Selection stage.

Syntax Description:

ANYCASE	Ignore case when comparing.
MIXED	The delimited string is to be used as a mask to test for all ones, mixed, or all zero bits. Only bit positions that correspond to one bits in the mask are tested; bit positions corresponding to zero bits in the mask are ignored. The string must not be null. Records are discarded if the delimited string satisfies the condition somewhere within the specified ranges.
ONES	
ZEROS	
	ANYOF cannot be specified with one of these keywords.
ANYOF	The delimited string specifies an enumerated set of characters rather than a string of characters. <i>nlocate</i> discards records that contain at least one of the enumerated characters within the specified input ranges.

No input range, a single input range, or one to ten input ranges in parentheses can be specified. The default is to search the complete input record.

The characters to search for are specified as a delimited string. A null string is assumed when the delimited string is omitted.

Operation: *nlocate* copies records that have no occurrence of the specified string within any specified input range (or that are shorter than the beginning of the input range that is first in the record) to the primary output stream, or discards them if the primary output stream is not connected. Thus, *nlocate* always selects null records. *nlocate* discards records in which the specified string occurs in a specified input range or copies them to the secondary output stream if it is connected.

A null string matches any record. In this case, records that are selected are shorter than the first position of the input range closest to the beginning of the record. This is used to select records shorter than a given length; “*nlocate 4|*” selects records of length 3 or less. Records of a particular length can be selected with a cascade of *nlocate* and *locate*.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *nlocate* strictly does not delay the record.

Commit Level: *nlocate* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *nlocate* terminates when it discovers that no output stream is connected.

Converse Operation: *locate*.

See Also: *nfind*.

Examples: To select null records:

...| *nlocate |*...

Notes:

1. Use a cascade of *nlocate* filters when looking for records not containing several strings.
2. *notlocate* is a synonym for *nlocate*.
3. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
4. Specifying MIXED and a mask that contains less than two one bits in any one byte will cause all records to be selected.

noeofback—Pass Records and Ignore End-of-file on Output

noeofback passes records as long as its output stream is connected. When its output is at end-of-file, it consumes the remaining input. Thus, *noeofback* propagates end-of-file forwards, but not backwards.



Type: Filter.

Record Delay: *noeofback* strictly does not delay the record.

Examples: To ensure that all records up to a target are consumed:

```
'callpipe (end ?)',
  '?*:', /* Input records */
  '|t: locate' target, /* Look for target */
  '|g: gate', /* Shut the door when it is there */
  '?t:', /* Records not the target */
  '|g:', /* Until the door gets shut */
  '|noeofback', /* Don't propagate EOF back */
  '|*:' /* Pass along */
```


If *noeofback* were omitted, end-of-file would propagate backwards through the *gate* stage and the subroutine would terminate before the first record containing the target. This, in turn, could cause the caller to malfunction.

Notes:

1. *noteofback* is a synonym for *noeofback*.

not—Run Stage with Output Streams Inverted

not runs the stage specified (most often a selection stage) with its output streams inverted. The primary output stream from the stage is connected to the secondary output stream from *not* (if it is defined). The secondary output stream from the stage is connected to the primary output stream from *not*. The stage must support two output streams.



Type: Control.

not

Syntax Description: Specify the name of the selection stage to run and its argument string.

Operation: The specified stage is run in a subroutine pipeline.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. The primary input stream is connected to the primary input stream of the stage. The primary output stream from the stage is connected to *not*'s secondary output stream if one is defined. The secondary output stream from the stage is connected to *not*'s primary output stream.

Record Delay: *not* does not add delay.

Commit Level: *not* starts on commit level -2. It verifies that the secondary input stream is not connected. *not* does not commit to 0; the stage called must do so.

Premature Termination: *not* terminates when the called stage terminates.

Examples:

To select the part of a record after the first period:

```
pipe literal abc.def | not chop . | console
►.def
►Ready;
```

Using *not* can simplify the pipeline topology when the primary output stream of a stage is not needed. The following outputs the unique records without the need to sort:

```
pipe literal x a b z b x y|split|not lookup autoadd | console
►x
►a
►b
►z
►y
►Ready;
```

Notes:

1. When both output streams are connected and the stage that is subject to *not* produces a record on both streams for each input record, the output records are produced in the reverse order of how they would be produced without the *not* qualifier.

For example, *not synchronise* and *not chop* both write a record to the secondary output stream before they write one to the primary output stream.

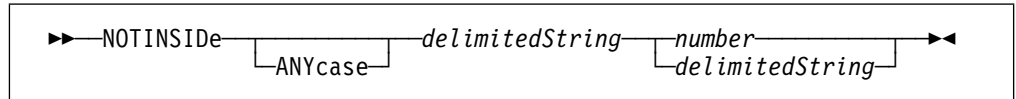
In general, dropping *not* into an existing pipeline in front of *synchronise* or similar can lead to a stall. It can lead to unexpected output when using stages that are sensitive to timing, such as *juxtapose*.

2. Many selection stages there is already a built-in that performs the inverse selection. For example, use *nfind* instead of *not find*.
3. The argument string to *not* is passed through the pipeline specification parser only once (when the scanner processes the *not* stage), unlike the argument strings for *append* and *preface*.
4. End-of-file is propagated from the streams of *not* to the corresponding stream of the specified selection stage.

Return Codes: If *not* finds no errors, the return code is the one received from the selection stage.

notinside—Select Records Not between Labels

notinside discards groups of records whose first record follows a record that begins with a specified string. The end of each group can be specified by a count of records to discard, or as a string that must be at the beginning of the first record after the group.



Type: Selection stage.

Syntax Description: A keyword is optional. Two arguments are required. The first one is a delimited string. The second argument is a number or a delimited string. The number must be zero or positive.

Operation: *notinside* discards groups of records or copies them to the secondary output stream if it is connected. Each group begins with the record after one that matches the first specified string. When the second argument is a number, the group has as many records as specified (or it extends to end-of-file). When the second argument is a string, the group ends with the record before the next record that matches the second specified string (or at end-of-file).

When ANYCASE is specified, *notinside* compares fields without regard to case. By default, case is respected. *notinside* copies records before, between, and after the selected groups to the primary output stream, or discards them if the primary output stream is not connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *notinside* strictly does not delay the record.

Commit Level: *notinside* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *notinside* terminates when it discovers that no output stream is connected.

Converse Operation: *inside*.

See Also: *between* and *outside*.

Examples: To remove lines inside example tags, while retaining the tags:

```
... | notinside /:xmp./ /:exmp./ | ...
```

Notes:

1. *ninside* is a synonym for *notinside*.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

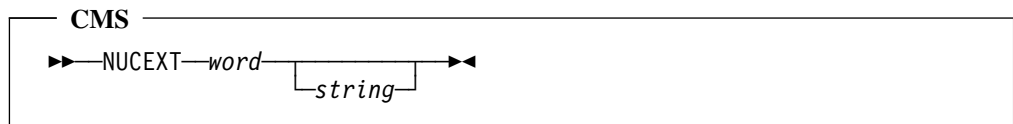
! 3. *pick* can do what *notinside* does and much more.

nucext—Call a Nucleus Extension

nucext resolves an entry point from a nucleus extension. The entry point can be:

- An executable machine instruction (not X'00'), which is then invoked as a stage.
- A program descriptor, which describes the stage to run.
- A pipeline command, which is run as a subroutine pipeline.
- An entry point table, from which the first word of the argument string is resolved.
- A look up routine, which resolves the first word of the argument string.

nucext is often used to test a compiled REXX program or an Assembler program before it is generated into a filter package.



Type: Arcane look up routine.

Syntax Description: Leading blanks are ignored; trailing blanks are significant. A word is required; additional arguments are allowed. The first word is the name of a nucleus extension to invoke. The name is translated to upper case if no extension is found with the name specified.

Operation: The optional string is passed to the program as the argument string.

Record Delay: *nucext* does not read or write records. The delay depends on the program being run.

Notes:

1. The nucleus extension is invoked as a filter with BAL. When the entry point is an executable instruction (not X'00'), general register 2 points to the SCBLOCK describing the nucleus extension. The nucleus extension is invoked enabled and in user key irrespective of the flags in the SCBLOCK; this is likely to cause a protection exception for a SYSTEM nucleus extension.
2. The program must be capable of being invoked in 31-bit addressing mode.
3. The nucleus extension must not use the CMSRET macro to return.
4. The nucleus extension must be able to distinguish between invocations from *CMS Pipelines* and invocations as a CMS command.
5. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

optcdj—Generate Table Reference Character (TRC)

optcdj makes records that have been prepared for a line printer suitable for printing on an IBM 3800, an IBM 3820, or a similar device. It inserts Table Reference Characters (which specify the font to use) after the carriage control character in the first position of each record. The input stream is assumed to contain descriptor records produced by *overstr*.



Type: Filter.

Syntax Description: A word is optional. If specified, it must be eight characters listing the TRCs for the eight categories of output listed below. The valid TRCs are 0, 1, 2, and 3; this is not enforced by *optcdj*. The default is '00001111'.

Operation: Input records with the two rightmost bits on in the carriage control character (B'xxxx xx11') represent immediate carriage movement; the carriage control character is passed to the output; additional data in the record are discarded. Records with data (that is, carriage control of the form B'xxxx xx01') that are not preceded by a descriptor record are assumed to contain plain characters; the third byte of the argument string (or the default '0') is inserted after the carriage control character, and the record is copied to the output (that is, a descriptor of X'02' is assumed for each non-blank column of an input record that is not preceded by a descriptor record).

Descriptor records and their accompanying data records are processed to assign a Table Reference Character to each position, using the descriptor value as an index into the argument string. Records are written for each Table Reference Character required. Line(s) with underscores are written for positions with underscored characters. The last record written for an input record has the carriage control character from the input record; other record(s) have X'01' carriage control (write no space).

Input Record Format: X'00' in the first column indicates a descriptor record in the format produced by *overstr*. Each column of the descriptor record specifies the highlighting and underscoring of the corresponding column in the data record that follows the descriptor record. These descriptor values are used:

- X'00' The position is blank.
- X'01' The position contains an underscore. (An underscored blank.)
- X'02' The position contains a character that is neither blank nor underscore.
- X'03' The position contains an underscored character.
- X'04' The position contains a highlighted blank.
- X'05' The position contains a highlighted underscore.
- X'06' The position contains a highlighted (overprinted) character.
- X'07' The position contains a highlighted and underscored character.

Records without X'00' in column 1 must begin with a machine carriage control character; data are from column 2 onward. Records that are not preceded by a descriptor record are neither underscored nor highlighted (though they can contain underscore characters).

Output Record Format: Descriptor records are not written to the output. Multiple output records are written for an input record that is preceded by a descriptor record. The first position is a machine carriage control character (it is never zero). The second position is a

outside

Table Reference Character. Data to print begin in column 3; the record extends to the last position requiring that particular TRC. Blanks (X'40') indicate columns that are not subject to the TRC for the record.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *optcdj* delays records that contain X'00' in column 1. It is unspecified if it delays other records. Application must not rely on *optcdj* to delay the other records.

Premature Termination: *optcdj* terminates when it discovers that its output stream is not connected.

See Also: *c14to38* and *overstr*.

Examples: To print a document formatted for an IBM 1403 on an IBM 3800 printer or an all points addressable (APA) printer under control of Print Services Facility (PSF):

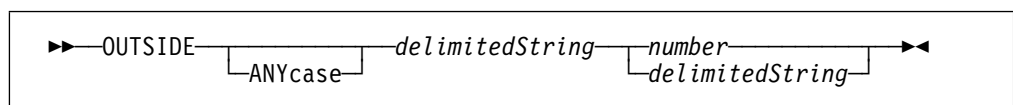
```
cp spool 00e fcb s8 char it12 ib12
cp tag dev 00e mvs system 0 OPTCD=J
pipe < $doc script | c14to38 | overstr | optcdj | printmc
cp close 00e
```

Notes:

1. Input records are truncated after 256 bytes without indication of error.
2. *optcdj* owes its name to the z/OS JCL option to indicate that a file contains Table Reference Characters: OPTCD=J.

outside—Select Records Not between Labels

outside discards groups of records whose first record begins with a specified string. The end of each group can be specified by a count of records to discard, or as a string that must be at the beginning of the last record.



Type: Selection stage.

Syntax Description: A keyword is optional. Two arguments are required. The first one is a delimited string. The second argument is a number or a delimited string. The number must be 2 or larger.

Operation: *outside* discards groups of records or copies them to the secondary output stream if it is connected. Each group begins with a record that matches the first specified string. When the second argument is a number, the group has as many records as specified (or it extends to end-of-file). When the second argument is a string, the group ends with the next record that matches the second specified string (or at end-of-file).

When ANYCASE is specified, *outside* compares fields without regard to case. By default, case is respected. *outside* copies records before, between, and after the selected groups to the primary output stream, or discards them if the primary output stream is not connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *outside* strictly does not delay the record.

Commit Level: *outside* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *outside* terminates when it discovers that no output stream is connected.

Converse Operation: *between*.

See Also: *inside* and *notin*.

Examples: To remove two lines of heading on each page from a report file with ASA carriage control in the first column.

```
...| mctoasa | outside /1/ 2 |...
```

Notes:

- 1. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
- 2. *pick* can do what *outside* does and much more.

outstore—Unload a File from a storage Buffer

outstore writes the contents of buffered files created by *instore* into the pipeline.



Type: Arcane filter.

Syntax Description:

```
ALET           The file is extracted from the specified data space.
```

ALET must be specified when *outstore* is first in the pipeline; no operands are allowed when *outstore* is not first in a pipeline.

Operation: When ALET is specified, the file is extracted from the specified data space.

Otherwise input records are read and the file is extracted from those descriptors. Such descriptors may or may not indicate that the file is in a data space.

Input Record Format: Records describe a file which is written to the output a record at a time. The input must be in the format produced by *instore*.

Streams Used: Records are read from the primary input stream and written to the primary output stream.

Record Delay: *outstore* does not delay the last record written for an input record.

overlay

Commit Level: *outstore* starts on commit level -2. It verifies the contents of the specified data space and then commits to level 0.

Premature Termination: *outstore* terminates when it discovers that its output stream is not connected.

Converse Operation: *instore*.

Examples: To reverse the order of the lines of a file:

```
...| instore reverse | outstore |...
```

overlay—Overlay Data from Input Streams

overlay combines a record from each connected input stream into a single output record. Each position in the output record contains the character from the highest numbered input stream where the corresponding position is present and not equal to the specified pad character.



Type: Gateway.

Syntax Description: A single character or two hex digits is optional to specify the pad character; the default is the blank character.

Streams Used: Records are read from all defined and connected input streams beginning with stream 0; output is written to the primary output stream only.

Record Delay: An input record is consumed as soon as it has been loaded into the output buffer; no input record is held while the output record is being written. Thus, *overlay* has the potential to delay one record on all input streams.

Commit Level: *overlay* starts on commit level -2. It verifies that the primary output stream is the only connected output stream and then commits to level 0.

Premature Termination: *overlay* terminates when it discovers that its primary output stream is not connected.

See Also: *spec* and *synchronise*.

Examples: To flush the contents of lines left and right, with a forward slash marking the point to insert blanks:

```
pipe (end ?) literal the left/the right | c: chop before / | ...
... o: overlay | console ? c: | spec 2-* 1.50 right | o:
▶the left                               the right
▶Ready;
```

Note that the pipeline does not stall, because *overlay* can delay the record on the primary input stream.

The following shows how characters from the record on the primary input stream are replaced by non-blank characters from the secondary input stream, but are retained at positions where the record on the secondary input stream has a blank character.

```
pipe (end ?) strliteral /abc fgh/ | o:overlay | ...
... console ? strliteral /1 34 789/ | o:
▶1b34 f789
▶Ready;
```

overstr—Process Overstruck Lines

overstr combines overstruck lines and creates descriptor records that specify the underscoring and highlighting of the data. The output is suitable for processing by *buildscr* and *optcdj*, in preparation for being displayed on a 3270 terminal or printed on an IBM 3820 printer, or similar.

►►—OVERSTR—◄◄

Type: Arcane filter.

Operation: A set of overstruck lines is merged into a single data record preceded by a descriptor record. In the merged data record, each position contains the character from the last line in the set of overprinted lines where the corresponding position is neither blank nor an underscore. A character position is considered overprinted when the position is neither blank nor underscore in two or more records of the set.

Lines not part of a set of overstruck lines are copied to the output without inspection. It is verified that all records have valid machine carriage control characters.

Input Record Format: The input records have machine carriage control characters in the first column. The input can contain sets of records that would be overstruck (printed on the same line) when sent to a line printer. A set of overstruck lines consists of one or more lines with X'01' carriage control (write no space) followed by a line with some other carriage control character.

Output Record Format: X'00' in the first column indicates a descriptor record. Each column of the descriptor record specifies the highlighting and underscoring of the corresponding column in the data record that follows the descriptor record. These descriptor values are used:

- X'00' The position is blank.
- X'01' The position contains an underscore. (An underscored blank.)
- X'02' The position contains a character that is neither blank nor underscore.
- X'03' The position contains an underscored character.
- X'04' Cannot occur.
- X'05' The position contains a highlighted underscore.
- X'06' The position contains a highlighted (overprinted) character.
- X'07' The position contains a highlighted and underscored character.

Each descriptor record is followed by a data record. The first character in a data record is the machine carriage control character from the last record of the corresponding set of overstruck input records. Data are from column 2 onward.

pack

Data records that are not preceded by a descriptor record contain no underscored or highlighted data (though they can contain underscore characters).

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *overstr* delays records that contain X'01' in column 1. It is unspecified if it delays other records. Application must not rely on *overstr* to delay the other records.

Premature Termination: *overstr* terminates when it discovers that its output stream is not connected.

See Also: *c14to38*, *optcdj*, *buildscr*, and *xpndhi*.

Examples: To print a document formatted for an IBM 1403 on an IBM 3800 printer or an all points addressable (APA) printer under control of Print Services Facility (PSF):

```
cp spool 00e fcb s8 char it12 ib12
cp tag dev 00e mvs system 0 OPTCD=J
pipe < $doc script | c14to38 | overstr | optcdj | printmc
cp close 00e
```

Notes:

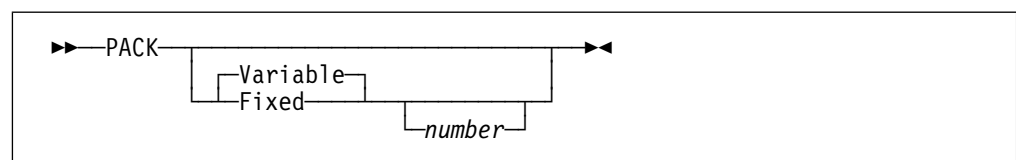
1. Use *asatomc* to convert from ASA to machine carriage control.
2. Use *c14to38* prior to *overstr* to change overstrikes of different characters to a single character.
3. *overstr* is designed to process the output from *c14to38* and deliver its output (possibly via *xpndhi*) to *buildscr* and *optcdj*.
4. For compatibility with the past, *delover* invokes this subroutine pipeline:

```
'callpipe *: | overstr | nfind' '00'x || '|' *:'
```

The resulting records have more data than the (rather naive) original *delover*.
5. No output record has X'01' carriage control.

pack—Pack Records as Done by XEDIT and COPYFILE

pack writes 1024-byte records containing an encoded version of the input data. The output has fewer bytes than the input when there are many runs of repeated characters.



Type: Filter.

Syntax Description: Two arguments are optional, a keyword and a number. They specify the file format and the maximum record length for the input. The default is variable record format with infinite record length. When FIXED is specified, the default record length is the length of the first input record.

Operation: The file is packed in 1024-byte records in the format used by COPYFILE and XEDIT. The last record is padded with binary zeros. When packing fixed record format, all input records must have the same length, which must be equal to the second argument, if it is specified.

The first record of a packed file contains an indication of the record length of the file; this can be set based on the first input record when packing a fixed file. When a record length is specified for a variable length file, no input record may be longer than this length.

When VARIABLE is specified without a maximum record length (or no arguments are specified) and the packed file contains more than one record, *pack* is not able to build a correct first record at the time it must be written to the output stream. Instead, *pack* writes a record that indicates an infinite record length (2G-1). The correct first record for the file is written to the secondary output stream after the entire file has been processed. The secondary output stream should be connected to the secondary input stream of *disk* to write the packed file to disk properly.

Warning: When the output of *pack* with variable record format is written to a file and the first record is not written correctly, the file can not be unpacked with XEDIT or COPYFILE (which give a message to the effect that the file is too big). *pack* can still unpack such a file.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null input records are discarded. Output is written to the primary output stream. One record is written to the secondary output stream when a variable format file is packed, if an explicit record length is not specified and more than one record is written to the primary output stream. The primary output stream is severed before *pack* writes to the secondary output stream.

Record Delay: *pack* delays input records as required to build an output record. The delay is unspecified.

Commit Level: *pack* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *pack* terminates when it discovers that its primary output stream is not connected.

Converse Operation: *unpack*.

Examples:

To pack a file that does not have records longer than 256:

```
pipe < some file|pack v 256|> packed file a fixed
```

If you do not know an upper limit on the record length, more care is required if the resultant file is larger than one disk block and you wish to read it with the CMS COPYFILE command or load it into XEDIT:

```
pipe (end ?)< some file|p: pack|d: > packed file a fixed?p:|d:
```

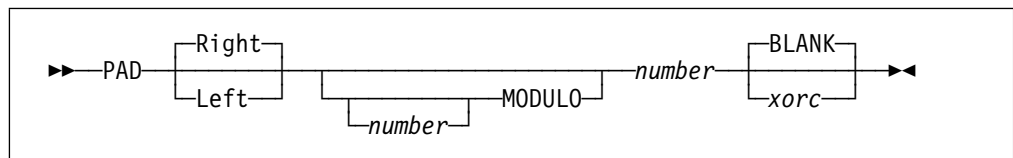
You must connect the secondary output stream from *pack* to the secondary input stream of *>* to write the correct first record at end-of-file, when the length of the longest record is known.

Notes:

1. When writing a packed file to disk, always specify fixed record format to the *disk* driver writing the file, even when the unpacked file is variable record format. XEDIT does not recognise a packed file when the record format is V.
2. *pack* can produce more bytes of output than it reads input. This is likely to happen when the data has random characters, for example in an encrypted file or a module file.

pad—Expand Short Records

pad expands records that are shorter than a specified length; a specified pad character is added to the beginning or the end of the record, as specified by a keyword. Records that are longer than the specified minimum length are passed on unmodified.



Type: Filter.

Syntax Description: Two keywords are optional. At least one number is required.

LEFT	Pad the record on the left. Thus, the text in the output record is aligned to the right.
RIGHT	Pad the record on the right. That is, add padding to the end of the record.
<i>number</i>	If it is specified, the first number must be smaller than the second. This number is called the offset. It must be followed by the keyword MODULO. The default offset is zero.
MODULO	The following number specifies not the absolute record length, but the modulo. The output record length is padded to the smallest multiple of the modulo plus the offset that is equal to or larger than the length of the input record.
<i>number</i>	Specify the minimum record length when MODULO is omitted. The number must be zero or positive. When MODULO is specified, the number must be positive.
<i>xorc</i>	The pad character is optional after the number; it may be specified as a single character or a two-character hex code. The default pad character is the blank.

By default, padding is on the right with blank characters.

Record Delay: *pad* strictly does not delay the record.

Premature Termination: *pad* terminates when it discovers that its output stream is not connected.

Converse Operation: *strip*.

Examples: To create a fixed record format file from console input:

```
pipe console | pad 80 | chop 80 | > input file a fixed
```

parcel—Parcel Input Stream Into Records

parcel treats its primary input stream as a stream of bytes, which is parcelled into records of the length specified by numbers read from the secondary input stream.



Type: Filter.

Operation: The byte stream on the primary input stream is reformatted into records of lengths as specified by the contents of the secondary input stream. If the secondary output stream becomes disconnected, *parcel* acts like *take n BYTES*, where the *n* is the sum of the numbers read from the secondary input stream. That is, at end-of-file of the secondary input stream, any remaining record from the primary input stream is passed to the secondary output stream and the primary input stream is then shorted to the secondary output stream.

Input Record Format: Blank-delimited numbers.

Streams Used: The secondary input stream must be defined and connected. The secondary output stream is optional.

Record Delay: *parcel* has the potential to delay one record. Output records that contain data from a single input record are not delayed, but are written as a burst. If an output record contains precisely the data in an input record, it is strictly not delayed. When both the records on the two input streams are emptied at the same time (that is, the last number in a record from the secondary input stream results in an output record that ends with the last byte of the record on the primary input stream), the record on the primary input stream is consumed before the one on the secondary input stream.

Commit Level: *parcel* starts on commit level -2. It verifies that the secondary output stream is not connected and then commits to level 0.

Premature Termination: *parcel* terminates when it discovers that its primary output stream is not connected. It also stops when the primary input stream reaches end-of-file or the secondary input stream reaches end-of-file and the secondary output stream is not connected. Message 72 is issued when the primary input stream reaches end-of-file before the secondary input stream if there is an unfinished record for the primary output stream; the last record read from the secondary input stream is not consumed. When the secondary input stream reaches end-of-file and the secondary output stream is not connected, *parcel* terminates immediately; the last record read from the primary input stream is not consumed when any data from it remain to be passed to the output.

See Also: *fblock*.

Examples: To generate a Christmas tree:

pause

```
xmastree
▶      *
▶      ***
▶      *****
▶      *******
▶      *********
▶      *
▶      *
▶      *****
▶Ready;

/* Produce a Christmas tree.                                     */
Signal on novalue
Address COMMAND
'PIPE (end ? name XMASTREE)',
  '?strliteral /*/',      /* Get a star                               */
  '|dup *',              /* Infinite supply                               */
  '|p: parcel',          /* Parcel them out                               */
  '|spec 1-* 1.20 centre', /* Centre                                         */
  '|console',
  '?literal 1 3 5 7 9 1 1 5', /* "magic" numbers                               */
  '|p:'
Exit RC
```

Notes:

1. Use *fblock* to format a byte stream into records of equal length.

pause—Signal a Pause Event

If the pipeline specification has been issued by *runpipe* EVENTS, *pause* signals a pause event for each input record. This event may be recognised by the stage that processes the output from the *runpipe* stage. The input record is then passed to the output.



Type: Arcane device driver.

Placement: *pause* must not be a first stage.

Operation: A pause event is signalled for each input record when the pipeline was issued by *runpipe* EVENTS. The record is then passed unmodified to the output.

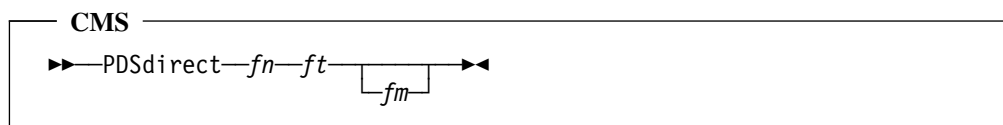
pause shorts the input to the output when it is running in a pipeline set that was not issued by *runpipe* EVENTS.

Record Delay: *pause* strictly does not delay the record.

Premature Termination: *pause* terminates when it discovers that its output stream is not connected.

pdsdirect—Write Directory Information from a CMS Simulated Partitioned Data Set

pdsdirect writes the first record and directory records from a CMS library on a minidisk or in a Shared File System (SFS) directory that has been accessed with a mode letter. The file must exist.



Type: Arcane device driver.

Placement: *pdsdirect* must be a first stage.

Syntax Description: Specify as blank-delimited words the file name and the file type of the file to be read. A file mode or an asterisk is optional; the default is to search all modes. If the file does not exist with the file name and the file type as entered, the file name and the file type are translated to upper case and the search is retried.

Operation: The file is closed before *pdsdirect* terminates.

Output Record Format: The first record written is record 1 of the file; it contains information about the type, position, and size of the PDS directory. The following records are the directory of the simulated PDS. The format of the directory records depends on the particular type of library.

Premature Termination: *pdsdirect* terminates when it discovers that its output stream is not connected.

See Also: *listpds* and *members*.

Examples: To write a list of members in a library:

```

/* MAPPDS REXX: Map a new-format MACLIB/TXTLIB */
signal on novalue
parse arg fn ft fm .
'callpipe (name MAPPDS)',
  |pdsdirect' fn ft fm ,           /* Read directory           */
  |drop 1',                       /* Drop pointer record      */
  |fblock 16',                    /* Deblock                  */
  |nfind' '00'x ,                 /* Deleted entries          */
  |spec 1.8 1 13.4 c2d next',     /* Format entry              */
  |*:'                             /* Output                   */
exit RC
  
```

Notes:

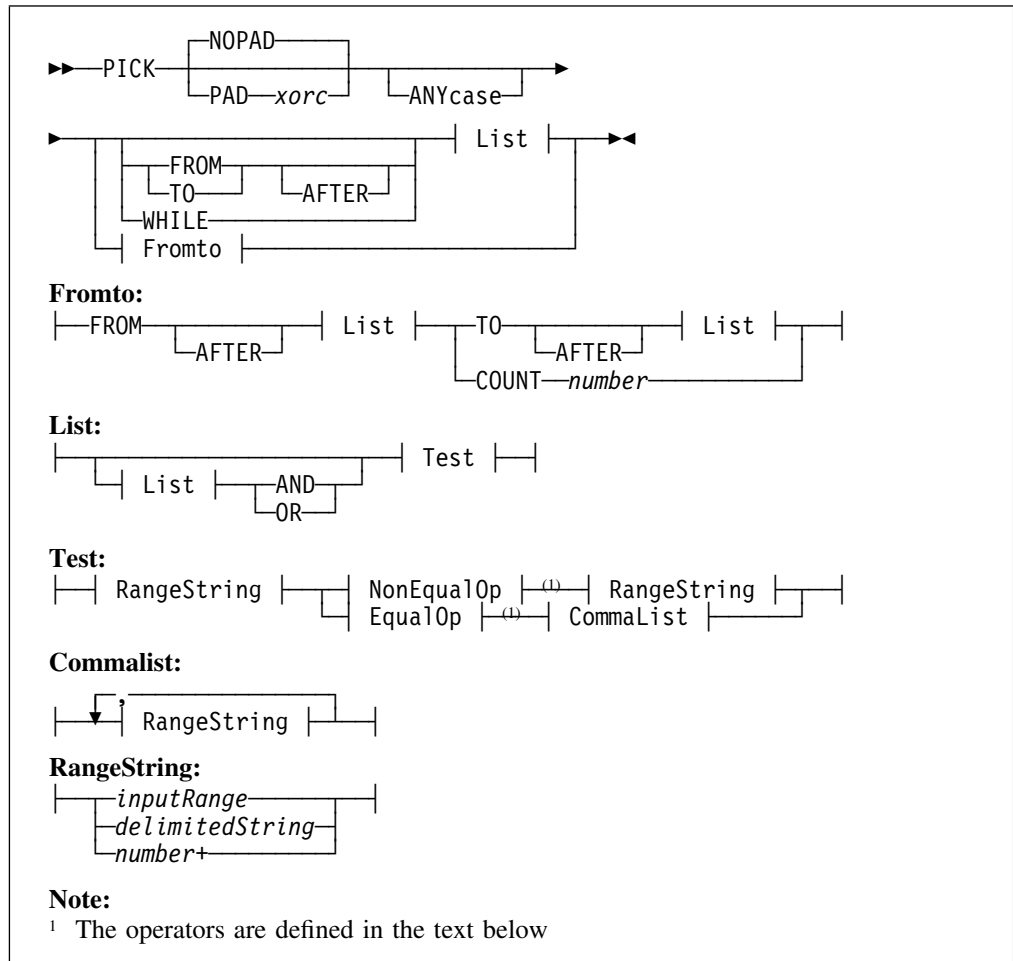
1. Use *listpds* rather than *pdsdirect* to obtain list of members in a PDS, one member to a record.
2. Use *members* to obtain members of simulated partitioned data sets that are fixed record format and record length 80, for instance TXTLIB and MACLIB.
3. *pdsdirect* reads CMS files; it does not read the directory of a PDS on an OS volume.

pick

pick—Select Lines that Satisfy a Relation

pick compares fields in the record with constants or other fields. The comparison can be of character data or numbers. It selects the record if the comparisons satisfy specified relations. It discards the record if the relation does not hold.

pick can also partition the file, selecting records from one that satisfies particular relations to one that satisfies other particular relations, or for a number of records.



Type: Selection stage.

Syntax Description:

AFTER Optional immediately after FROM or TO. When specified, the selection action is performed after the matching record; when omitted it is performed before the matching record.

AND Match only when both comparisons match. AND has higher precedence than OR. Specify parentheses to group OR comparisons. You may use an ampersand (&) or even two (&&) instead of the keyword.

ANYCASE Case is to be ignored when comparing strings; the default is to respect case.

:	COUNT	Specified in conjunction with FROM. When a record is matched by the FROM clause, the specified number of records are passed to the primary output stream.
:		
:		
:	FROM	Select from the matching record. Records before the matching record are rejected. When specified without TO or COUNT, the balance of the file is selected. Otherwise records are selected for the specified count or up to one that is selected by the TO clause.
:		
:		
:	NOPAD	Do not pad the shorter operand when comparing strings. When PAD is omitted, the unpaired positions in the longer string are considered to compare high. (Thus, the shorter string is logically extended with a value that compares low against X'00'.)
:		
:	OR	Match when either comparison matches. You may use a vertical bar () if you use a different stage separator, escape it with an escape character, or use two () instead of the keyword.
:		
:	PAD	Specify the padding character to use when comparing strings. The shorter of the two strings is extended on the right with the specified pad character for purposes of comparison.
:		
:	WHILE	Select the first part of the file until, but not including, a record does not match. Reject the balance of the file.
:		

A comparison is specified by two operands with an operator in between. The left side can contain one operand only. The right hand side may contain a list of operands separated by commas for the two equal operators; other operators accept one operand only. Each operand may be:

- An *inputRange*. This can designate a manifest constant, in which case the comparison must be numeric.
- A *delimitedString*, which specifies a constant, being it a number for numeric comparison or a true string.
- A *number* followed immediately by a plus (+) without blanks. (Automatic field length.) The number specifies the beginning column. The length used will be the smaller of the length of the other operand and the length of the rest of the record from the specified column. You can specify only one of the operands as an auto field length. The left operand cannot have auto field length when the right hand side is a list of operands separated by commas.

The relational operators for strings are adopted from the REXX “strict” relational operators. *pick* ignores the member type in character comparison.

:	==	Strictly equal. The two strings must be equal byte for byte except for case folding and padding. The right hand side may be specified as a list of operands separated by commas; the relation holds when at least one of the operands compare equal with the first operand. The inverse of \neq .
:		
:	\neq	Not strictly equal. The inverse of $==$.
:	<<	Strictly less than. After a (possibly null) run of bytes that are the same in the two strings, the left string must contain a character that is lower in the collating sequence than the corresponding character in the right hand string. The inverse of $>>=$.
:		
:	<<=	Strictly less than or equal. The inverse of $>>$.

pick

>>	Strictly greater than. After a (possibly null) run of bytes that are the same in the two strings, the left string must contain a character that is higher in the collating sequence than the corresponding character in the right hand string. The inverse of <<=.
>>=	Strictly greater than or equal. The inverse of <<.
IN	Match when all characters of the first operand are present in the second one or the first operand is null.
NOTIN	Inverse of IN.

When using relational operators for numeric comparisons of data that have no type associated, the fields or constants must conform to the syntax described in “Floating point Numbers” on page 741. For typed members of structures the input members are converted automatically for types D, F, P, R, and U. Note that literal numbers must also be specified as a *delimitedString* to distinguish them from columns. The numeric relational operators are:

=	Equal. The two numbers must have the same sign and be exactly equal. The right hand side may be specified as a list of operands separated by commas; the relation holds when at least one of the operands compare equal with the first operand. The inverse of \neq .
\neq	Not equal. The inverse of =.
<	Less than. The inverse of \geq .
\leq	Less than or equal. The inverse of >.
>	Greater than. The inverse of \leq .
\geq	Greater than or equal. The inverse of <.

Operation: *pick* copies records that satisfy the specified relation to the primary output stream, or discards them if the primary output stream is not connected. It discards records that do not satisfy the relation or copies them to the secondary output stream if it is connected.

- When neither FROM, TO, nor WHILE is specified, *pick* selects lines that match the list of comparisons and discards those that do not.
- When FROM, TO, or WHILE is specified *pick* partitions the file at the first line that matches (or not):
 - FROM rejects the part of the file up to the first matching record. When AFTER is specified, the matching record is rejected; otherwise it is selected. The remainder of the file is selected.
 - TO selects the part of the file up to the first matching record. When AFTER is specified, the matching record is selected; otherwise it is rejected. The remainder of the file is rejected.
 - WHILE selects records until, but not including, the first one that does not match; it then rejects the remainder of the file.
- When COUNT or TO is specified with FROM, *pick* discards records up to the first record that is matched by the FROM list. It then selects either the number of records specified by COUNT or up to the next record that matches the TO list.

: When TO is specified without AFTER, the matching record is not written immediately;
 : instead the FROM clause is retested against the record to see whether it starts a new
 : range to be selected. This has effect when the TO clause is a subset of the FROM
 : clause.

: Having selected the specified records, *pick* then goes back to rejecting records until
 : another one is matched by the FROM list.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *pick* strictly does not delay the record.

: **Commit Level:** *pick* starts on commit level -2. It parses the argument list, then verifies that the secondary input stream is not connected, and then commits to level 0.

Premature Termination: *pick* terminates when it discovers that no output stream is connected.

See Also: *all*.

Examples: Assuming that input records contain a timestamp in the first four columns, select the records processed earlier than 8 a.m:

```
... | pick 1.4 << "0800" | ...
```

: To select records within an interval; for example, to select the records timestamped from 8
 : am. to but not including 4 p.m., (15:59 on a 24-hour clock):

```
... | pick 1.4 >>= "0800" and 1.4 <=<= "1559" | ...
```

: This could also have been achieved with a cascade of two *pick* stages.

: To select records outside an interval:

```
... | pick 1.4 << "0800" or 1.4 >>= "1600" | ...
```

: This cannot be accomplished with a cascade of *pick* stages; a multistream topology would
 : be required.

pick can compare two fields in a record; for example, to select records that represent files that need updating. Assuming that input records contain one ISO-format timestamp (YYYYMMDDHHMMSS) in columns 23 to 36 and another one in columns 57 to 70, select those records where the first timestamp is later than the second:

```
... | pick 23.14 >> 57.14 | ...
```

To select records where the second word is equal to the fourth word. This also selects records that contain one word only because both operands would then be null.

```
... | pick word 2 == word 4 | ...
```

Two ways to select records that contain at least two words:

```
... | pick word 2 -== // | ...
```

```
: ... | locate word 2 | ...
```

: To select records where the second word contains a number that is greater than the number
 : in columns 37 to 41:

pick

```
... | pick word 2 > 37.5 | ...
```

To select records where the second word is greater than the constant 37.5:

```
... | pick word 2 > /37.5/ | ...
```

Consider a stacked COPY file where members are separated by a *COPY record. To select all members beginning with A:

```
...|pick from 1+ == /*COPY / and substr 1 of w2 == /A/  
to 1+ == /*COPY /|...
```

Notes:

1. \== and /== are synonyms for !=. \= and /= are synonyms for =. The not sign is often mapped by terminal emulators to the caret (^).
2. You can specify a literal as both the first and the second string. All records are then either selected or rejected, depending on the static relation between the two constants.
3. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
4. Input ranges for numeric compare are converted in the same way done by *spec* for a field that has a field identifier specified.
5. The typical use of IN would be to test whether a column contains one of a set of characters.
6. The only sensible use of parentheses is to enclose OR items in conjunction with AND, but redundant parentheses are accepted.
7. The following two invocations of *pick* select the same set of records:

```
...|pick w6 == /take/ & ( 1 == /a/ or 1 == /b/ or 1 == /c/ )|...  
...|pick w6 == /take/ and 1 == /a/, /b/, /c/|...
```

The motivation for this construct is laziness, but with a complex left hand input range there may be some measurable performance increase with large files.

8. As an example, these are equivalent:

```
...|between /abc/ /def/|...  
...|pick from 1+ == /abc/ to after 1+ == /def/|...
```

As are these:

```
...|inside /abc/ /def/|...  
...|pick from after 1+ == /abc/ to 1+ == /def/|...
```

Other variations of AFTER are available that cannot be implemented with any of the *between* family of selection stages.

9. The *between* family of stages is made redundant by the enhancements introduced in *CMS Pipelines* 1.1.11/1D, but they are scanned faster because their syntax is simpler. Runtime performance should be equivalent.
10. An *inputRange* must be terminated with a blank (unlike a *delimitedString*).

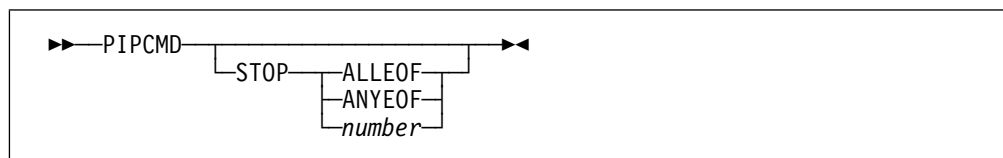
```
...|pick 1 == 2, 3|...
```

Will give a rather misleading diagnostic. The following is syntactically valid, but unlikely to do what you want, because it will not select anything as the second operand is six bytes literal delimited by the two “2” digits and no padding is specified:

```
...|pick 1 == 2, 3, 12|...
```

pipcmd—Issue Pipeline Commands

pipcmd issues input records as pipeline commands. Typically, the input consists of CALLPIPE pipeline commands that are built from data by a *spec* stage earlier in the pipeline.



Type: Control.

Syntax Description:

STOP	Inspect the status of the output streams after each input line has been processed.
ANYEOF	Terminate as soon as any stream is at end-of-file.
ALLEOF	Terminate as soon as all streams are at end-of-file.
<i>number</i>	Terminate when the specified number of streams are at end-of-file.

Operation: With no keywords specified, the equivalent REXX program is:

```

/* PIPCMD */
retcode=0
trace off
do until RC=0
  'peekto command'          /* Read a line          */
  if RC<0 then leave        /* Stall?              */
  if RC>0 then exit retcode /* EOF?                */
  ' command'                /* Issue the pipeline command */
  if RC<0 then leave        /* Real trouble         */
  retcode=max(retcode, RC)  /* Make worst return code */
  'readto'                  /* Discard the input line */
end
exit RC

```

Note that an input line is not consumed before the corresponding command is complete. A record is discarded by the READTO if the command consumes records from the primary input stream.

Input Record Format: Input lines may contain any pipeline command described in Chapter 25, “Pipeline Commands” on page 750 except for BEGOUTPUT, GETRANGE, NOCOMMIT, PEEKTO, READTO, SCANRANGE, SCANSTRING, and STREAMSTATE ALL. The most useful one is no doubt CALLPIPE.

Streams Used: *pipcmd* reads commands from the primary input stream; it does not write output, but the commands may connect to any defined stream. The input record is consumed after the command completes.

A pipeline command should not refer to the primary input stream. If it does, the first line it sees is the one issued as the command; a line is discarded from the primary input stream

when the command completes (this line is the line containing the command when the primary input stream has not been read by the command).

Premature Termination: *pipcmd* terminates as soon as a negative return code is received. It also terminates when STOP is specified and the specified number of output streams are at end-of-file. The corresponding input record is consumed.

See Also: *getfiles* and *runpipe*.

Examples: To process the contents of the files whose names match the pattern specified by the argument string, in this case looking for the string "Dana":

```

/*-----*/
/* Get the contents of files */
/*-----*/
parse arg fn ft fm .
'PIPE (name PIPCMD)',
  'command LISTFILE' fn ft fm,
  '|spec', /* Generate subroutine */
  '/callpipe (stagesep ?) / 1', /* Command */
  '/?< / next 1-* next', /* Prefix read of file */
  '/?spec ,/ next w1 next /, 1.8 1-* 10/ next', /* Add fn */
  '/?*:/ next', /* Connect to caller's stream */
  '|pipcmd',
  '|locate 10-* /Dana/',
  '|cons'

```

For each file found by LISTFILE, the first *spec* stage builds a subroutine pipeline with a question mark as the stage separator. The subroutine pipeline reads the file (<) and puts the file name in columns one through eight (the second *spec*). Thus, the first *spec* stage generates another one of the form *spec ,fn, 1.8 1-* 10*. The CALLPIPE pipeline commands are issued by *pipcmd* to read the files and prefix the file name to each record of a file. The *locate* stage selects records containing the string. The result is displayed on the terminal.

A variation on this approach is showing with *juxtapose*.

To test whether a particular program is built in:

```
pipe literal resolve insert | pipcmd
```

The program is not built in when the return code is zero. (A nonzero return code represents the entry point address in storage.)

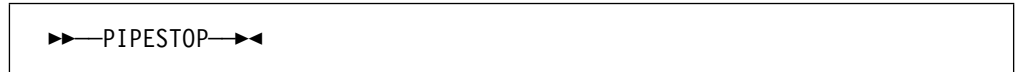
The RESOLVE option in *filterpack* provides a more flexible way to test for built-in programs.

Return Codes: When a negative return code is received on a pipeline command, the return code from *pipcmd* is that negative return code. When the return code is zero or positive, all input records have been processed; the return code is the maximum of the return codes received.

pipestop—Terminate Stages Waiting for an External Event

pipestop posts with code X'3F' all ECBs in the pipeline set that are being waited on by other stages. This forces the stages to terminate.

pipestop also sets an indication in the pipeline set that will prevent further waiting for external events. This action is irreversible.



Type: Arcane control.

Placement: *pipestop* must not be a first stage.

Operation: When an input record arrives, *pipestop* posts all ECBs that are being waited upon by other stages. The completion code indicates that the stages should terminate. *pipestop* then passes the input record to the output if it is connected.

Streams Used: *pipestop* passes the input to the output.

Record Delay: *pipestop* strictly does not delay the record.

Examples: To terminate processing after one minute:

```
pipe (end ?) ... ? literal +60 | delay | pipestop
```

Note that the first pipeline (not shown) should also have a *pipestop* stage to terminate the delay set up here.

Alternatively, you can use *gate* to shut down the timer:

```
pipe (end ?) ... | g: gate ? literal +60 | delay | g: | pipestop
```

Either pass a record to *gate* when you wish to turn off the delay, or use *hole* to make *gate* wait for end-of-file:

```
pipe (end ?) ... | take last | g: gate ? literal +60 | delay | g: | pipestop
```

Notes:

1. To terminate one particular waiting stage rather than all waiting stages, design the topology such that a *gate* cuts off the input or output of the waiting stage. The asynchronous stage terminates when it finds the input or output disconnected.

polish—Reverse Polish Expression Parser

Parse expressions and generate the reverse polish list of operations that must be performed to evaluate each expression.



Type: Filter.

Syntax Description: A keyword is required.

polish

:	HEXADECIMAL	Parse according to the hexadecimal grammar, which can implement a hexadecimal pocket calculator.
:	ASSEMBLER	Parse according to a grammar that is compatible with Assembler
:	ASM	expressions, such as, the first operand of the EQU instruction.

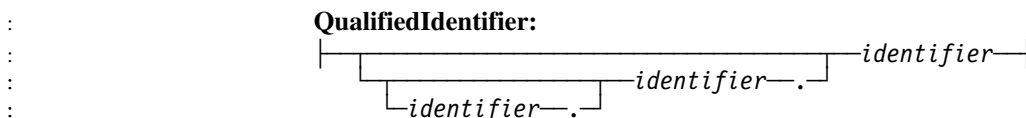
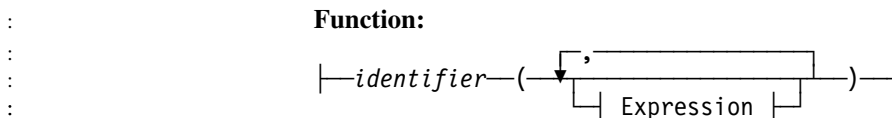
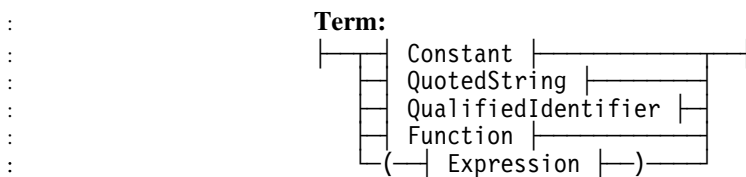
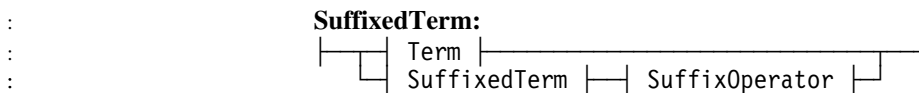
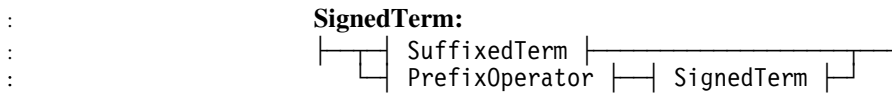
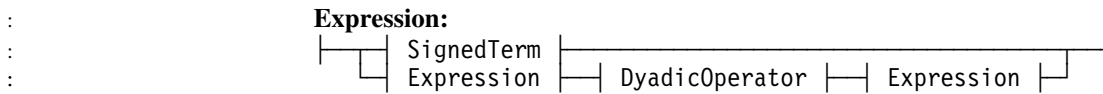
Operation: *polish* parses each input line according to the specified grammar. It then writes to the primary output stream a list of actions to perform. The list is ended by a null line. Output lines contain a word specifying the action type and other data that an evaluator would need to evaluate the expression.

The expression result will be the single item on the evaluation stack when the evaluator reaches a null input record (assuming, of course, that a correct evaluator is supplied).

Input Record Format: For the hexadecimal parser:

Blanks are ignored between terms, but you cannot have blanks in constants, identifiers, or the composite operators (*//*). In particular, blanks are allowed between an identifier and the left parenthesis that opens the argument list, and even in conjunction with periods that qualify an identifier.

Input lines should conform to this syntax:



Dyadic Operators: The following table lists the dyadic operators in order of increasing precedence; that is, the last one (binary AND) binds closer than any of the others. All operators in a row have the same precedence. All dyadic operators are left associative, that is, $a+b+c$ is equivalent to $(a+b)+c$

Figure 393. Dyadic Operators

Operator	Type	Suggested interpretation
+ -	Additive	Addition and subtraction
* / % //	Multiplicative	Multiplication, division, integer division, remainder.
	OR	Bitwise OR
&	AND	Bitwise AND

Prefix Operators: The prefix operators are plus (+) and minus (-). They bind tighter than dyadic operators, but not as close as suffix operators.

Figure 394. Prefix Operators

Operator	Type	Suggested interpretation
+ -	Prefix	Prefix plus might cause truncation to a 31-bit number.

Suffix Operators: The suffix operator is the question mark (?). Nothing binds closer than the question mark (unless you consider the period in a qualified identifier to be an operator; it binds even closer).

Figure 395. Suffix Operators

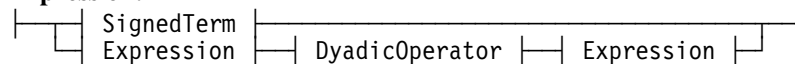
Operator	Type	Suggested interpretation
?	Suffix	Indirection, for example to fetch the contents of a specified location in virtual storage.

For the Assembler parser:

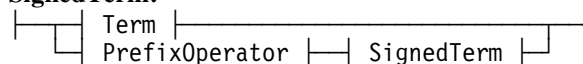
The input record can contain blanks only in character self-defining terms. The parser is caseless, that is, upper case and lower case are equivalent as far as the parser is concerned, but the output respects the case of the input.

Input lines should conform to this syntax:

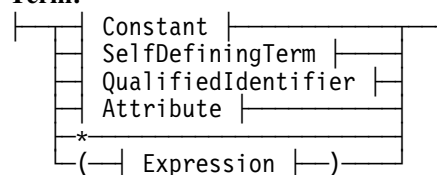
Expression:



SignedTerm:

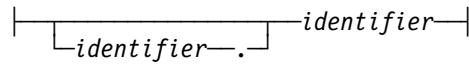


Term:

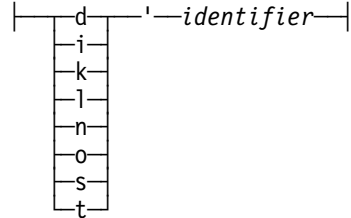


QualifiedIdentifier:

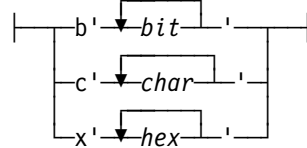
polish



Attribute:



SelfDefiningTerm:



Dyadic Operators: The following table lists the dyadic operators in order of increasing precedence; that is, the last ones (multiply and divide) binds closer than any of the others. All operators in a row have the same precedence. All dyadic operators are left associative, that is, $a+b+c$ is equivalent to $(a+b)+c$

Figure 396. Dyadic Operators

Operator	Type	Suggested interpretation
+ -	Additive	Addition and subtraction
* /	Multiplicative	Multiplication, integer division.

Prefix Operators: The prefix operators are plus (+) and minus (-). They bind tighter than dyadic operators.

Figure 397. Prefix Operators

Operator	Type	Suggested interpretation
+ -	Prefix	Prefix minus should return the negative of the term.

Output Record Format: The output records contain actions to perform. The first word contains a keyword that specifies the action; additional data is present depending on the particular keyword.

Figure 398 (Page 1 of 2). Hexadecimal Parser

Keyword	Word 2+	Suggested Evaluator Action
constant	The second word contains the numeric constant ("abc" is a numeric constant with the HEX parser).	Push the literal value on the evaluation stack.

Figure 398 (Page 2 of 2). Hexadecimal Parser

Keyword	Word 2+	Suggested Evaluator Action
string	The second word to the end of the line contains a quoted string, including the quotes. Both single and double quotes are supported for the string begin character.	Push the literal value on the evaluation stack.
identifier	An identifier (that does not also parse as a hexadecimal constant for the HEX parser). For HEX parsers, the character set is the same as for the High Level Assembler. Case is preserved in identifiers, but your evaluator may, of course, decide to fold the names.	Push the value of the identifier on the evaluation stack.
qualifier	A qualified identifier. Word two through the end of the line contains qualifiers and the identifier without periods in the reverse order they appeared in the expression.	Resolve the qualified identifier and push its value on the evaluation stack. The two levels of qualifier may be interpreted as module and control section.
function	Call a function. The second word contains the function name. The third word contains the number of arguments to pop off the stack.	Pop the specified number of values from the evaluation stack, perform the function, and push the result.
monadic	A monadic operator. The second word contains the operator. This includes prefix and suffix operators; the parser has determined the correct order to apply them; thus the evaluator need not distinguish between the two types.	Apply the operator to the top of the evaluation stack and replace it with the result.
binary	A dyadic operator. The second word contains the operator.	Pop two values from the evaluation stack, apply the operator to the values, and push the result.

Figure 399 (Page 1 of 2). Assembler Parser

Keyword	Word 2-3	Suggested Evaluator Action
constant	The second word contains the numeric constant. Self-defining terms are converted by the parser to their signed decimal equivalent.	Push the literal value on the evaluation stack.
identifier	An identifier. Case preserved in identifiers, but your evaluator should be caseless. The line for a qualified identifier contains the qualifier as the third word.	Push the value of the identifier on the evaluation stack.
attribute	The attribute character, a blank, and the identifier.	Push the value of the attribute of the identifier on the evaluation stack.
loctr	*	Push the current location counter on the evaluation stack.

polish

Figure 399 (Page 2 of 2). Assembler Parser

Keyword	Word 2-3	Suggested Evaluator Action
monadic	A monadic operator. The second word contains the operator.	Apply the operator to the top of the evaluation stack.
binary	A dyadic operator. The second word contains the operator.	Pop two values from the evaluation stack, apply the operator to the values, and push the result. The result of division by 0 is 0 irrespective of the dividend.

Record Delay: *polish* does not delay the record.

Premature Termination: *polish* terminates when it discovers that its output stream is not connected.

Examples:

A fairly simple example:

```
pipe literal max ( g1+10, deadbeaf&30)|polish hex | console
▶identifier      g1
▶constant       10
▶binary         +
▶constant       deadbeaf
▶constant       30
▶binary         &
▶function       max 2
▶
▶Ready;
```

Using qualifiers:

```
pipe literal . v - z+x.y?|polish hex | console
▶qualifier      v
▶identifier     z
▶binary        -
▶qualifier     y x
▶monadic       ?
▶binary        +
▶
▶Ready;
```

The expression below might compute the address of the field pointed to by the fullword twelve bytes after the address pointed to by fullword addressed by the contents of register four (depending on the evaluator, of course):

$(r4?+c)?$

If register 4 contains X'10' and storage location X'10' contains X'20', the result could be the contents of location X'2C'.

The Assembler parser is simpler in comparison:

```

:      pipe strliteral /*-dsect/ | polish assemble | console
:      ▶loctr          *
:      ▶identifier    dsect
:      ▶binary        -
:      ▶
:      ▶Ready;

```

Notes:

1. A sample evaluator for the hexadecimal parser can be found at <http://vm.marist.edu/%7epipeline/evalx.exec>
2. You cannot modify the underlying grammar. In particular, the precedence of the operators cannot be changed. You may implement a subset of the grammar by, for example, rejecting particular operators at evaluation time.
3. ASM is a synonym for ASSEMBLER.

predselect—Control Destructive Test of Records

predselect is designed to assist in building complex selection stages where input records may be tested destructively. That is, input records may be changed or rejected. Testing must not delay the record. The primary input stream should contain the original file; derivative records are fed to the secondary input stream or the tertiary input stream.



Type: Gateway.

Operation: *predselect* reads records from whichever input stream has one available. It stores the last record read from the primary input stream in a buffer (replacing any previous content) and then consumes the record to release the producer (which is typically *fanout*). Records read from the secondary input stream and the tertiary input stream are discarded; they merely control which stream should receive the stored record read from the primary input stream. A record on the secondary input stream causes the buffered record to be written to the primary output stream; a record on the tertiary input stream causes the buffered record to be written to the secondary output stream. Once the buffered record is written, subsequent input records on the secondary input stream or the tertiary input stream are discarded until a record is read from the primary input stream.

When a record arrives on the primary input stream without intervening records on the two other input streams, the previous record is in effect discarded.

Streams Used: Two streams must be defined; up to three streams may be defined. *predselect* propagates end-of-file between the secondary input stream and the primary output stream; and it propagates end-of-file between the tertiary input stream and the secondary output stream.

Record Delay: *predselect* has the potential to delay one record. *predselect* delays the record by any delay the record may have incurred before it reaches the secondary input stream or the tertiary input stream; *predselect* does not add delay as long as all input streams are connected and a record on the primary input stream is followed by a record on the secondary input stream or the tertiary input stream.

Commit Level: *predselect* starts on commit level -2. It verifies that the tertiary output stream is not connected and then commits to level 0.

Premature Termination: *predselect* terminates when it discovers that no output stream is connected. It terminates as soon as its primary input stream is severed.

See Also: *juxtapose*.

Examples: FINDANY REXX performs caseless selection using a *find* stage:

```
/* FINDANY REXX -- FIND ignoring case */
Signal on novalue
parse upper arg args
'maxstream output'
If RC=0
  Then out1='' /* No secondary output */
  Else out1='|*.output.1:' /* Route rejected records here */
'callpipe (end ? name ANYCASE)',
'|*:', /* Input here */
'|o: fanout', /* Keep a pristine copy */
'|p: predselect', /* Control selection */
'|*.output.0:', /* Those selected */
'?o:', /* Copy of the input record */
'|xlate upper', /* Make it uppercase */
'|s: find' args ||, /* Perform the selection */
'|p:', /* Pass to control */
out1, /* maybe write output */
'?s:', /* Rejected records */
'|p:' /* To rejection input */
exit RC
```

```
pipe literal another one | literal A record | findany a | console
▶A record
▶another one
▶Ready;
```

The argument string to the *find* stage is made upper case, because it operates on a copy of the file that is upper case.

Note that the output from the example would be the same if the last pipeline (feeding the secondary output stream from the selection stage to the tertiary input stream to *predselect*) were omitted. Including this pipeline ensures that the subroutine as a whole never delays the record; without this connection, discarded records would be delayed until the next record became available on the primary input stream.

This example is contrived because *strfind* ANYCASE performs the same function cheaper.

preface—Put Output from a Device Driver before Data on the Primary Input Stream

preface runs a device driver to generate output which is passed to *preface*'s output; *preface* then passes all its input records to the output.

▶▶—PREFACE—*string*—◀◀

Type: Control.

Syntax Description: The argument string is normally a single stage, but any pipeline specification that can be suffixed by a connector (`|*:`) is acceptable (see usage note 1).

Operation: The string is issued as a subroutine pipeline with `CALLPIPE`, using the default stage separator (`|`), double quotes as the escape character (`"`), and the backward slash as the end character (`\`). The beginning of the pipeline is unconnected. The end of the pipeline is connected to *preface*'s primary output stream. (Do not write an explicit connector.) The input records are passed to the output after the `CALLPIPE` pipeline command has completed.

In the subroutine pipeline, device drivers that reference REXX variables (*rexxvars*, *stem*, *var*, and *varload*) reach the `EXECCOMM` environments in effect for *preface*.

Streams Used: The string that specifies the subroutine pipeline can refer to all defined streams except for the primary output stream (which will be connected to the end of the subroutine pipeline by *preface*). The primary input stream is shorted to the primary output stream when the subroutine pipeline ends.

Record Delay: *preface* delays the input file by the number of records that are prefaced. These records are written before the input file is read.

Commit Level: *preface* starts on commit level -1. The subroutine pipeline must commit to 0 if it generates output.

Premature Termination: *preface* terminates if the `CALLPIPE` pipeline command gives a nonzero return code. The subroutine pipeline may or may not have committed to 0 before the error is discovered.

See Also: *append* and *literal*.

Examples: To put the contents of a variable before the stream being built:

```
...| preface var firstline |...
```

Notes:

1. The argument string may contain stage separators and other special characters. Be sure that these are processed in the right place. The argument string is passed through the pipeline specification parser twice, first when the pipeline containing the *preface* stage is set up, and secondly when the argument string is issued as a subroutine pipeline. The two example pipelines below show ways to preface a subroutine pipeline consisting of more than one stage. In both cases, the *split* stage is part of the subroutine pipeline and, thus, splits only the record produced by the second literal stage:

```
pipe literal c d e| preface literal a b || split | console
▶a
▶b
▶c d e
▶Ready;
```

```
pipe (sep ?) literal c d e? preface literal a b | split ? console
▶a
▶b
▶c d e
▶Ready;
```

In the first example, the stage separator that is to be passed to the subroutine pipeline is self-escaped in the main pipeline. In the second example, the stage separator for the

main pipeline is a question mark; thus, no special treatment is required to pass the stage separator (|) to *preface*.

Now consider how to specify a vertical bar as part of an argument to the subroutine pipeline (in both cases, the variable data has the value abc|def):

```
pipe var data | preface var data || split ||| | console
▶abc
▶def
▶abc|def
▶Ready;
```

```
pipe (stagesep ?) var data ? preface var data | split || ? console
▶abc
▶def
▶abc|def
▶Ready;
```

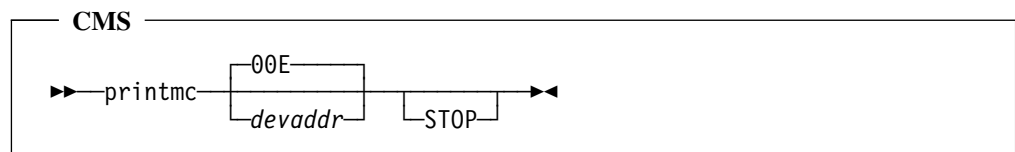
In the first example, the stage separator that should be recognised in the subroutine pipeline is self-escaped; to get the parameter (a single |) through the pipeline specification parser twice, it must be doubly self-escaped; that is, the four vertical bars become one when the argument is presented to *split*. In the second example, the main pipeline uses the question mark as its stage separator and thus no escape is required to pass the vertical bar to the subroutine pipeline; and a single self-escape suffices to get the vertical bar to *split*.

- Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: The return code is the return code from the CALLPIPE pipeline command. It may reflect errors in the argument string or trouble with the stage(s) in the pipeline.

printmc—Print Lines

printmc copies lines from the pipeline to a virtual printer. The lines must have machine carriage control characters.



Type: Device driver.

Placement: *printmc* must not be a first stage.

Syntax Description: Arguments are optional. Specify the device address of the virtual printer to write to if it is not the default 00E. The virtual device must be a unit record output printer device. The keyword STOP allows you to inspect the channel programs built by *printmc*.

Operation: The first byte of each record designates the CCW command code (machine carriage control character); it is inserted as the CCW command code. The remaining characters are identified for transport to SPOOL by the address and length fields of the CCW. A

single blank character is written if the input record has only the command code. Control and no operation CCWs can specify data; the data are written to the SPOOL file. X'5A' operation codes are supported, but other read commands are rejected with an error message; command codes are not otherwise inspected.

Records may be buffered by *printmc* to improve performance by writing more than one record with a single call to the host interface. A null input record causes *printmc* to flush the contents of the buffer into SPOOL, but the null record itself is not written to SPOOL. After the producing stage has written a null record, it is assured that *printmc* can close the unit record device without loss of data. Input lines are copied to the primary output stream, if it is connected.

printmc issues no CP commands; specifically, the virtual device is not closed.

The virtual Forms Control Buffer (FCB) for a virtual printer (the virtual carriage control tape) can be loaded by a CCW or the CP command LOADVFCB. The channel program is restarted after a channel 9 or 12 hole causes it to terminate; even so, such holes in the carriage tape should be avoided, because they serve no useful purpose; and they generate additional overhead.

Record Delay: *printmc* strictly does not delay the record.

Commit Level: *printmc* starts on commit level -2000000000. It ensures that the device is not already in use by another stage, allocates a buffer, and then commits to level 0.

See Also: *reader*, *punch*, and *uro*.

Examples: To print a file with carriage control:

```
pipe < pgm listing | asatmc | printmc
cp close 00e name pgm listing
```

To close the printer every 50 records:

```
'PIPE (end ? name PRINTMC.STAGE:36)',
  '?... ',
  '|o: fanout',          /* Get two copies          */
  '|i:faninany',       /* Merge with nulls       */
  '|printmc',          /* Print; nulls flush     */
  '?o:',               /* The records            */
  '|chop 0',           /* Make them null         */
  '|join 49',          /* Join 50 null records   */
  '|c: fanout',        /* Still a null record    */
  '|i:',               /* Send to printer        */
  '?c:',               /* Trigger record         */
  '|spec /CLOSE 00E/', /* Build command          */
  '|cp'                /* Issue it                */
```

The trick is to pass a null record to *printmc* to force it to flush the contents of its buffer into CP SPOOL before the device is closed.

Notes:

1. *printmc* has been tested with a virtual printer; its error recovery is unlikely to be adequate for a dedicated printer.

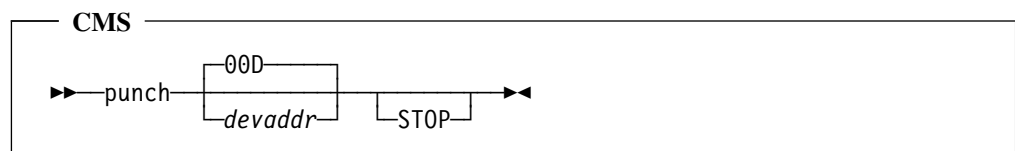
punch

2. Any output data can be written, including 3800 CCWs, but be aware that CP support depends on the virtual device type. For example, the maximum record length (including CCW operation code prefix) is 133 bytes on a virtual 1403.
3. STOP causes CP console function mode to be entered after each channel program has been given to CP. General register 2 points to the HCPSGIOP data area, from which information about the channel program can be extracted.

Make sure you SET RUN OFF when using this option. This function was written to help debug *printmc*, but it may also be useful to discover errors in input data.

punch—Punch Cards

punch copies lines from the pipeline to punched cards.



Type: Device driver.

Placement: *punch* must not be a first stage.

Syntax Description: Arguments are optional. Specify the device address of the virtual punch to write to if it is not the default 00D. The virtual device must be a unit record output punch device. The keyword STOP allows you to inspect the channel programs built by *punch*.

Operation: Each input record that is not null is written to the punch with the command write, feed, select stacker 2 (X'41').

Records may be buffered by *punch* to improve performance by writing more than one record with a single call to the host interface. A null input record causes *punch* to flush the contents of the buffer into SPOOL, but the null record itself is not written to SPOOL. After the producing stage has written a null record, it is assured that *punch* can close the unit record device without loss of data. Input lines are copied to the primary output stream, if it is connected. Any output data can be written, but CP truncates cards after 80 bytes without error indication.

punch issues no CP commands; specifically, the virtual device is not closed.

Record Delay: *punch* strictly does not delay the record.

Commit Level: *punch* starts on commit level -2000000000. It ensures that the device is not already in use by another stage, allocates a buffer, and then commits to level 0.

See Also: *reader*, *printmc*, and *uro*.

Examples: To punch a file without carriage control and no header record:

```
pipe < some file | punch
cp close d name some file
```

To close the punch every 50 records:

```
'PIPE (end ?)',
'|?... ',
'|o: fanout', /* Get two copies */
'|i:faninany', /* Merge with nulls */
'|punch', /* Punch; nulls flush */
'?o:', /* The records */
'|chop 0', /* Make them null */
'|join 49', /* Join 50 null records */
'|c: fanout', /* Still a null record */
'|i:', /* Send to punch */
'?c:', /* Trigger record */
'|spec /CLOSE 00D/', /* Build command */
'|cp' /* Issue it */
```

The trick is to pass a null record to *punch* to force it to flush the contents of its buffer into CP SPOOL before the device is closed.

Notes:

1. Use *uro* to create punch files that contain records having command code X'03' (no operation).
2. *punch* has been tested with a virtual card punch; its error recovery is unlikely to be adequate for a dedicated card punch.
3. STOP causes CP console function mode to be entered after each channel program has been given to CP. General register 2 points to the HCPSGIOP data area, from which information about the channel program can be extracted.

Make sure you SET RUN OFF when using this option. This function was written to help debug *punch*, but it may also be useful to discover errors in input data.

qpdecode—Decode to Quoted-printable Format

qpdecode decodes records according to the Multipurpose Internet Mail Extensions (MIME). *qpdecode* operates in the ASCII domain.

▶▶—QPDECODE—◀◀

Type: Filter.

Syntax Description:

Operation: Escape sequences of the form X'3dxxxx' are converted to the single character represented by the two encoded characters. Other characters are passed unchanged to the output record.

X'3d' at the end of the line means to splice with the following line.

The escape sequences are validated for being complete and correct ASCII (case is ignored, however). When this validation fails and no secondary output stream is defined, a message is issued and *qpdecode* exits. When the secondary output stream is defined, any partial record stored as a result of line splicing with a trailing equal sign is written to the primary output, the entire erroneous input record is passed to the secondary output and processing continues.

qpencode

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *qpdecode* does not delay the record. *qpdecode* does not delay the last record written for an input record.

Commit Level: *qpdecode* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *qpdecode* terminates when it discovers that any of its output streams is not connected.

Converse Operation: *qpencode*.

Notes:

1. *qpdecode* is concerned solely with decoding. More work is required to build a complete receiver for MIME encoded files.
2. If you are decoding mail, it is probable that the receiving mailer has “converted” the file to EBCDIC. You must convert it back to ASCII before it is passed to *qpdecode*. (And then back to EBCDIC.)
3. *qpdecode* validates only escape sequences. Thus, it passes characters that are not valid in an encoded file.
4. On z/VM releases before z/VM 6.4, *qpdecode* was not a built-in program. Some developers may have used a REXX stage that operates on data in EBCDIC rather than in ASCII. The application will be affected when *rex* was not used to specify use of the REXX stage. To exploit the new built-in *qpdecode* the REXX stage can be replaced by

```
.. | xlate from 1047 to 819 | qpdecode | xlate from 819 to 1047 | ..
```

Publications: (MIME) quoted-printable encoding format is defined in a range of RFCs starting with 2045.

qpencode—Encode to Quoted-printable Format

qpencode encodes records according to the Multipurpose Internet Mail Extensions (MIME). *qpencode* operates in the ASCII domain.

▶▶—QPENCODE—◀◀

Type: Filter.

Operation: Encode records according to the MIME quoted-printable encoding. For each input record, as many 76-byte output records are produced as required; the last record from a particular input record is in general shorter than 76 bytes.

In addition to the encoding mandated by the standard, *qpencode* escapes those special characters that are not codepage invariant in the EBCDIC domain. These are X'21222324' X'40', X'5b5c5d5e', X'60', and X'7b7c7d7e'.

Record Delay: *qpencode* does not delay the last record written for an input record.

Premature Termination: *qpencode* terminates when it discovers that no output stream is connected.

Converse Operation: *qpdecode*.

Notes:

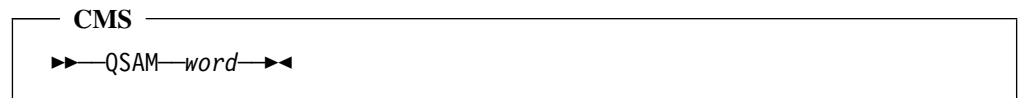
1. *qpencode* is concerned solely with encoding. More work is required to build a complete MIME encoded file.
2. On z/VM releases before z/VM 6.4, *qpencode* was not a built-in program. Some developers may have used a REXX stage that operates on data in EBCDIC rather than in ASCII. The application will be affected when *rex* was not used to specify use of the REXX stage. To exploit the new built-in *qpencode* the REXX stage can be replaced by


```
.. | xlate from 1047 to 819 | qpencode | xlate from 819 to 1047 | ..
```

Publications: (MIME) quoted-printable encoding format is defined in a range of RFCs starting with 2045.

qsam—Read or Write Physical Sequential Data Set through a DCB

qsam uses queued sequential processing to read or write a physical sequential data set.



Type: Device driver.

Warning: *qsam* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *qsam* is unintentionally run other than as a first stage. To use *qsam* to read data into the pipeline at a position that is not a first stage, specify *qsam* as the argument of an *append* or *preface* control. For example, `|append qsam ...|` appends the data produced by *qsam* to the data on the primary input stream.

Syntax Description: A word is required; it represents the DDNAME to use.

Operation: The data set is read when *qsam* is first in a pipeline; it is written when *qsam* is not first in a pipeline.

qsam generates record descriptor words and block descriptor words when it writes a data set in variable format (V, VB, VS, or VBS). Such record descriptor words are removed when *qsam* reads a data set. On CMS *qsam* does not support spanned records; it can write undefined record format data sets, but it cannot read them.

Record Delay: *qsam* strictly does not delay the record.

Commit Level: *qsam* starts on commit level -2000000000. It opens the data set and then commits to 0.

Premature Termination: When it is first in a pipeline, *qsam* terminates when it discovers that its output stream is not connected.

See Also: *disk*, *<*, *>*, *>>*, *members*, *pdsdirect*, *readpds*, and *writpds*.

Notes:

1. *qsam* can read sequential data sets from OS disks, but CMS does not support writing on OS disks. Use *disk* or one of its “almost synonyms” (<, >, or >>) to read and write CMS files.
2. *qsam* reads and writes a DCB, not necessarily a disk file; it does not investigate where the data set is allocated.
3. On CMS, a data definition must have been established with a FILEDEF for the specified DDNAME before the pipeline specification is issued.
4. Use the LABELDEF command in conjunction with FILEDEF to process standard labelled tapes.
5. To be compatible with the past, *qsam* is also shipped for TSO. It may work, but it is not supported.

query—Query CMS Pipelines

query obtains information from *CMS Pipelines*. The information is written to the output (if connected) or issued as a message.



Type: Service program.

Placement: *query* must be a first stage.

Syntax Description: A keyword is optional. The default is to display the version message.

Operation: A message is issued with the information requested when the primary output stream is not connected. Message 86 is issued to display the pipeline version; message 186 displays the message level; message 189 displays the list of messages issued; message 560 displays the pipeline level.

A line is written (no message is issued) when the primary output stream is connected.

Output Record Format: When the primary output stream is connected, a record is written in this format:

VERSION	The text for message 86, including ten characters prefix.
MSGLEVEL	Four bytes of binary data.
MSGLIST	44 bytes containing 11 items of four bytes each. The message number is in the first three bytes; the severity code is in the last one. When the message number is 999 or less, it is stored as three characters; when larger than 999 it is stored as a packed decimal number without sign. The last item corresponds to the last message issued; the first item corresponds to the message issued the least recently. Leftmost items are binary zeros when fewer than 11 messages have been issued.

LEVEL Four bytes of binary data. The version (B'0001') is stored in the first four bits. The release (B'0001') is stored in the next four bits. The modification level (B'00001100') is stored in the next eight bits. The last sixteen bits are a serial number for the particular build of the PIPELINE MODULE.

Commit Level: *query* starts on commit level -4. It commits to level 0 when the argument keyword is validated.

Examples: To display only the modification level of *CMS Pipelines*:

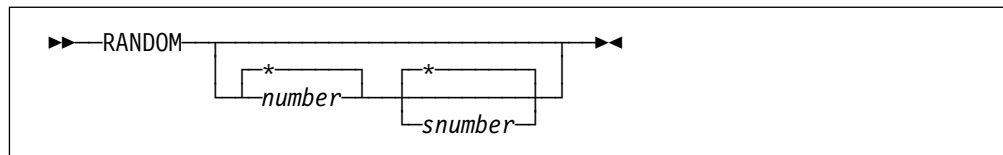
```
pipe query level | spec 2 c2d 1 | strip | console
▶12
▶Ready;
```

Notes:

1. The message number is added to the message list when *CMS Pipelines* issues a message, except for messages 1, 3, 4, 189, 192, 260, 278, and 836. These messages are informational to describe the conditions under which the previous message was issued. There is one list for all stages, pipelines, pipeline specifications, and pipeline sets.

random—Generate Pseudorandom Numbers

random writes output records four bytes long that contain binary pseudorandom values.



Type: Device driver.

Placement: *random* must be a first stage.

Syntax Description:

number Specify the modulus if you wish to restrict the values of the output numbers. The modulus must be positive; the output number is the remainder of the pseudorandom number after division by the modulus; the output numbers are positive. Specify an asterisk as a placeholder.

snumber Specify a number to be used as a seed for the sequence of numbers. If the seed is omitted or an asterisk is specified, a seed is obtained from the time-of-day clock.

Premature Termination: *random* terminates when it discovers that its output stream is not connected. *random* does not terminate normally.

Examples:

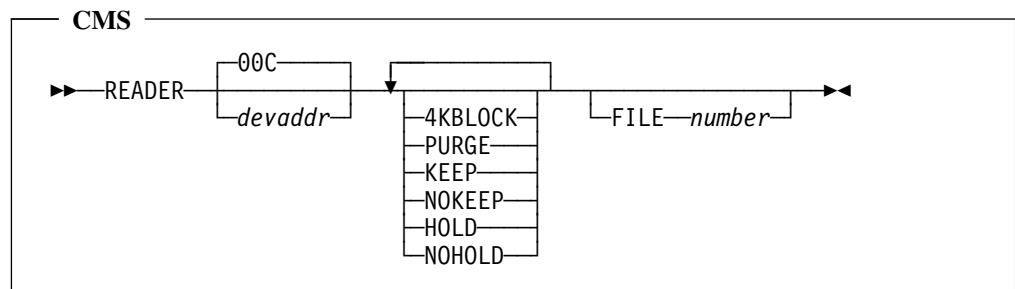
reader

```
pipe random | take 3 | spec 1-* c2x 1 | console
▶56A3DBDD
▶DBDD1B77
▶1B77EA03
▶Ready;
pipe random 7| take 3 | spec 1-* c2x 1 | console
▶00000000
▶00000000
▶00000006
▶Ready;
```

reader—Read from a Virtual Card Reader

reader reads a file from a virtual reader. The file can be a print file, a punch file, or a VMDUMP file. *reader* deblocks the file to individual records unless instructed to write the complete 4K SPOOL buffers.

Warning: By default *reader* does not change the SPOOL settings of the virtual reader it uses. Be sure to issue the CP command “spool reader hold” or specify the appropriate options on *reader* if you wish to retain a reader file after it has been read.



Type: Device driver.

Placement: *reader* must be a first stage.

Syntax Description: Arguments are optional.

Specify the device address of the virtual reader to read from, if it is not the default 00C. The virtual device must be a unit record reader device.

4KBLOCK specifies that each complete 4K buffer should be written to the pipeline without deblocking.

If any of the options PURGE KEEP NOKEEP HOLD NOHOLD are specified, they are added to the CLOSE command that closes the reader after the file has been read.

The number after the keyword FILE designates a particular reader file to be processed.

Operation: When FILE is specified, the file is selected for the reader before the first block is read. The file must not be in hold status; it must have a class that can be read by the reader in question. When FILE is omitted, CP selects the next available SPOOL file that is not held and is compatible with the reader.

Unless the 4KBLOCK option is specified or implied, the reader file is deblocked into records that contain the command code in the first position followed by the data. Trailing blanks are added if the SPOOL file contains the original length of the record. All CCWs are written

including control and no operation. Data chained sequences (which often span input blocks) are joined into one logical record.

: The reader is closed with the CP command “close” when processing completes without
: error; the reader is left open when *reader* terminates due to an error.

Commit Level: *reader* starts on commit level -2000000000. It determines that the device is not already in use by another stage, selects the file (if FILE is specified), and then commits to level 0.

Premature Termination: *reader* terminates when it discovers that its output stream is not connected.

See Also: *printmc*, *punch*, and *uro*.

Examples: A subroutine pipeline that deblocks a reader file that has been sent as a note or with the SENDFILE command:

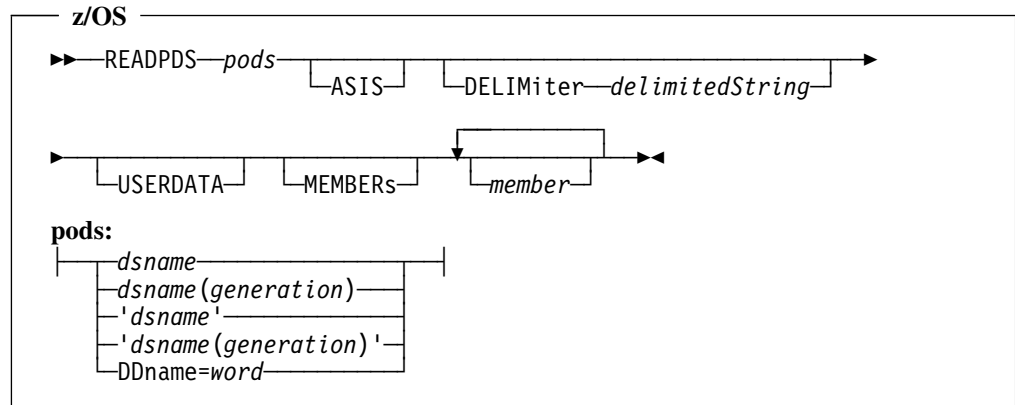
```
/* Now get the file */
'callpipe (name READER)',
  | reader ', /* Read cards */
  | strfind x41 ', /* Take only cards */
  | spec 2-* 1.80 ', /* Remove CCW code */
  | deblock netdata ', /* Get logical records */
  | strfind xc0 ', /* Take only data */
  | spec 2-* 1 ', /* Remove control byte */
  | *: ' /* Pass on */
```

Notes:

1. *reader* does not support an attached card reader.
2. 4KBLOCK must be specified to read a VMDUMP file; results are unpredictable if the option is omitted when reading a VMDUMP file.
3. *reader* cannot read CP dump files The references to VMDUMP above include CP dumps.
4. There are at least two no operation CCWs at the beginning of a file that has arrived through RSCS/Networking: the current tag and the original tag.
5. To retain the trailing blanks in the records of a SPOOL file that has been transmitted through a network, all nodes traversed in the network must store the original length of the record in the SPOOL file; once this information is lost, it cannot be regenerated.
6. Use the CP command “spool reader keep” or “change rdr nnnn keep” to put a reader file in user hold status.
7. Specifying KEEP puts the file in user hold status, whereas HOLD leaves the file so that it can be read again immediately. Using NOKEEP with a virtual reader that is spooled KEEP appears to leave the file in the reader.
8. *reader* terminates and leaves the SPOOL file open when it receives an error from CP. It closes the file when it receives end-of-file writing a record.

readpds—Read Members from a Partitioned Data Set

readpds reads members from a library into the pipeline. The member names may be specified as arguments, or they may be provided in input records, or both.



Type: Device driver.

Syntax Description:

- | | |
|-------------|---|
| <i>pods</i> | Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the DSNAME without quotes to have the prefix, if any, applied. Append parentheses containing a signed number to specify a relative generation of a data set that is a member of a generation data group. To read members of an already allocated data set, specify the keyword DDNAME= followed by the DDNAME already allocated. The minimum abbreviation is DD=. |
| ASIS | Use member names strictly as written. By default, a member name is translated to upper case if it is not found in the mixed case spelling. |
| DELIMITER | Specify the beginning of the delimiter record, which is written between members. The member name is appended to this string. |
| USERDATA | Append the user data field from the directory record to the delimiter record. The user data is unpacked to printable hexadecimal. |
| MEMBERS | The remaining words are names of members to be read. MEMBERS is assumed when a word is scanned that is not a recognised keyword. |

A blank-delimited list of member names is optional.

Operation: *readpds* first reads the contents of members (if any) specified in the argument string; it then continues with the members specified in input records.

Each member is looked up in the library directory. If the member does not exist as written and ASIS is omitted, the search is retried with the member name translated to upper case.

A delimiter record is written before each member, if DELIMITER is specified.

A null record is written after each member.

Diagnostic messages are issued for members that are not present in the library; the argument and all input records are processed before returning with return code 150 when one or more members is not found.

Input Record Format: Blank-delimited lists of members to read from the library.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *readpds* writes all output for an input record before consuming the input record.

Commit Level: *readpds* starts on commit level -2000000000. It opens the DCB and then commits to level 0.

Premature Termination: *readpds* terminates when it discovers that its output stream is not connected.

See Also: *listispcf* and *listpds*.

Examples: To read the member J from the PDS allocated to SYSEXEC:

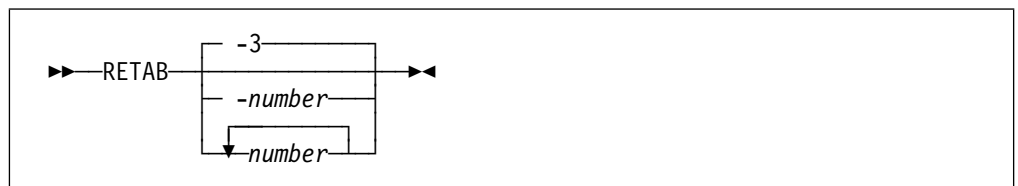
```
pipe literal j | readpds dd=sysexec | cons
▶/* REXX */ parse arg file
▶address link
▶'PIPE <' file '|menuct1 /'file'/ edit'
▶exit rc
▶READY
```

Notes:

1. *members* and *pdsread* are synonyms for *readpds*.

retab—Replace Runs of Blanks with Tabulate Characters

retab replaces runs of blanks in the record with tabulate characters.



Type: Filter.

Syntax Description: No arguments are required. A single negative number or a list of positive numbers may be specified.

A list of positive numbers enumerates the tab stops; the numbers may be in any order. The smallest number specifies where the left margin is; use 1 to put the left margin at the beginning of the record.

A negative number specifies a tab stop in column 1, and for each *n* columns.

The default is -3, which is equivalent to 1 4 7 ...

Operation: When a list of tab stops is used and the smallest number is not 1, the first columns of the record are discarded up to the column specified as the left margin.

Record Delay: *retab* strictly does not delay the record.

Premature Termination: *retab* terminates when it discovers that its output stream is not connected.

Converse Operation: *untab*.

Examples: To generate a file that will contain tabs:

```
pipe < ebcdic file|retab -8|block 4000 c term|xlate e2a|> unix file a
```

reverse—Reverse Contents of Records

reverse reverses the contents of each record so that the first character becomes the last, the last character becomes the first, the penultimate character becomes the second, the second character becomes the penultimate, and so on.

►—REVERSE—◄

Type: Filter.

Record Delay: *reverse* strictly does not delay the record.

Premature Termination: *reverse* terminates when it discovers that its output stream is not connected.

Converse Operation: *reverse*.

Examples:

```
pipe literal Hello, World! | reverse | console
► !dlroW ,olleH
►Ready;
```

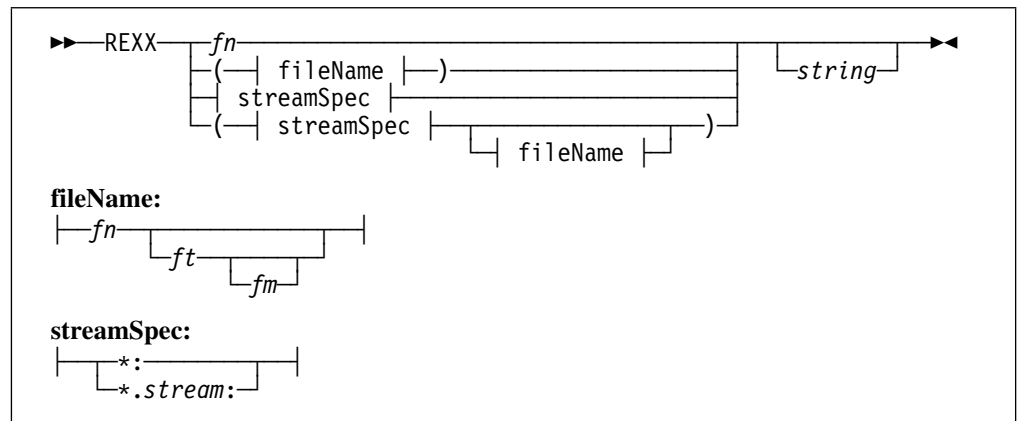
To find records that end in “xyz”:

```
... | reverse | find zyx| reverse | ...
```

Note that the argument to *find* is reversed because the contents of the record are reversed at the point where *find* is applied.

rexx—Run a REXX Program to Process Data

rexx runs a REXX program in the pipeline with access to input and output streams. *rexx* is implied when no built-in program is found with a particular name; specify *rexx* explicitly to bypass a built-in program, to run a program with a file type other than REXX, or to run the program from an input stream.



Type: Look up routine.

Syntax Description: Leading blanks are ignored; trailing blanks are significant. A word is required; it may be followed by a string. If the first non-blank character is a left parenthesis, up to three words for file name, type, and mode can be specified in parentheses. The default file type is REXX. On z/OS, the file type specifies the DDNAME of the library that contains the program. When the first word begins with an asterisk and ends with a colon, it specifies the input stream from where the REXX program is read; to set the file name to be used in the REXX source string, the stream specification followed by the file name (and optionally file type and file mode) must be specified in parentheses.

Operation: A REXX program runs as a pipeline filter. The string is passed to the program as its argument string. The default command environment processes pipeline commands, described in Chapter 25, “Pipeline Commands” on page 750. For a task-oriented guide, see Chapter 7, “Writing a REXX Program to Run in a Pipeline” on page 97 and “Using CALLPIPE to Run a Subroutine Pipeline” on page 103.

An EXEC can invoke itself as a filter. The seventh word of the source string is a question mark when the program runs as a filter on CMS; it is PIPE on z/OS.

If, on CMS, the program is not already loaded in storage by an explicit EXECLOAD, it is loaded with the EXECLOAD command before it is invoked. Concurrent invocations of a program use the same copy as long as the file is accessible and has the same timestamp; the program is removed from storage when the last concurrent invocation terminates. Compiled REXX programs (with option CEXEC) are invoked by direct branch to the compiler runtime environment. If the runtime environment is not installed as a nucleus extension, it is invoked with CMSCALL to make it initialise itself.

Streams Used: The REXX program can select any defined stream using the SELECT pipeline command; it can define additional streams with the ADDSTREAM pipeline command; it can determine the number of defined streams from the return code from the MAXSTREAM pipeline command.

Record Delay: *rexx* does not read or write records. The delay depends on the program being run.

Commit Level: *rexx* starts on commit level -1. When the program is read from an input stream, *rexx* commits to zero before reading the program. When the program is not read from an input stream, *rexx* commits to level 0 (unless the command NOCOMMIT has been issued) when the first I/O operation (OUTPUT, PEEKTO, or READTO pipeline commands) is

requested or the pipeline command SELECT ANYINPUT is issued. The program can issue COMMIT before doing I/O to test whether any other stage has returned with a nonzero return code on commit level -1.

Examples: An EXEC that invokes itself as a filter:

```
/* Hello, world. */
signal on novalue
parse source . . $fn $ft . . how .
If how='?'
    Then signal filter
        address command
            'PIPE rexx (' $fn $ft ')|console'
        exit RC
filter:
    'output Hello, world! (From a dual-path REXX.)'
    exit RC
```

```
hello2
▶Hello, world! (From a dual-path REXX.)
▶Ready;
```

To read the program from the primary input stream:

```
pipe literal /**/ 'output Hello, World!' | rexx *: | console
▶Hello, World!
▶Ready;
```

TESTALT EXEC reads the program from the secondary input stream:

```
/* TESTALT EXEC */
'PIPE (end ? name REXX)',
'|literal me Tarzan',
'|r: rexx *.1:',
'|console',
'?|literal /* */ 'output Hello, World!'; 'short"',
'|r:'
```

```
testalt
▶Hello, World!
▶me Tarzan
▶Ready;
```

Notes:

- ! 1. There is also a REXXCMD command to invoke a REXX program as subroutine rather than as a stage.
- ! 2. You need not use an explicit *rexx* to invoke a REXX program with file type REXX unless there is a built-in program with the same name; *CMS Pipelines* looks for a REXX program when it cannot resolve a filter in the built-in directories and attached filter packages.
- 3. Compiled REXX programs are supported on CMS. Programs can be compiled with the OBJECT option or the CEXEC option. The former programs are included in filter packages; the latter are run from disk or EXECLOADED.
- 4. A program that is used often should be EXECLOADED to improve performance.
- 5. *CMS Pipelines* installs an alternate EXEC interpreter in a slightly different way than CMS does: When there is no nucleus extension installed for the processor, it is called

to install itself. This is done by CMSCALL with a call type program (flag byte is zero) and register zero cleared to zeros. The interpreter should install itself (or its runtime routine) as a nucleus extension and return. *CMS Pipelines* then looks again for the runtime environment and branches to it. *CMS Pipelines* supports only EXECs requiring an alternate processor that installs itself as a nucleus extension.

6. On z/OS, REXX filters run in dedicated reentrant environments. Such environments cannot be merged with the TSO environment. Issue TSO commands with *command*, *tso*, or *subcom* TSO instead.

```
/* Issue a TSO command */
'callpipe command time'
```

7. The *rex* verb cannot be defaulted when the program is to be read from an input stream.

:
!
!
!
!
!
!
!

8. Remember that REXX on CMS resolves an external function call using the type of the program (for example, REXX when the function call is from a REXX filter). If you have a filter with the same name as an external function, the filter will be invoked rather than the corresponding EXEC. REXX allows for no way to avoid this. To call the EXEC program from a REXX filter, you can use the EXEC prefix in the function name:

```
/* F00 REXX */
signal on error
'peekto rec'
'output' 'EXEC F00'(rec)
error: return rc * (rc <> 12)
```

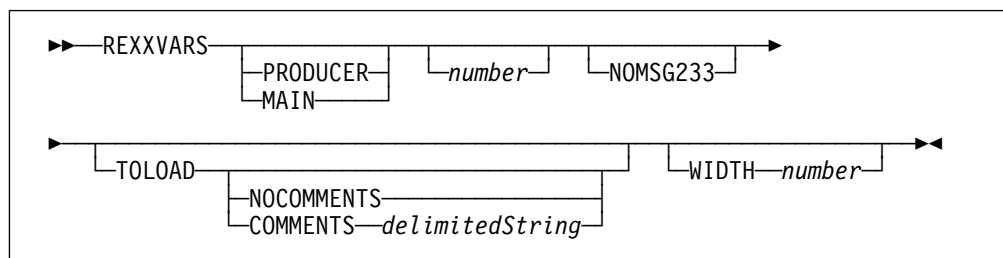
9. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: Unless it is a return code associated with trouble finding the REXX program to run, the return code is the one received from the REXX program.

rexvars—Retrieve Variables from a REXX or CLIST Variable Pool

rexvars writes the names and values of the currently exposed REXX variables (including the source string) into the pipeline. *rexvars* can retrieve variables from the current EXEC (or REXX pipeline program) or from one of its ancestors.

Warning: The output from *rexvars* must be buffered if other stages access the variable pool concurrently, for example to return results with *stem*.



Type: Device driver.

Placement: *rexxvars* must be a first stage.

Syntax Description: It is possible to access a REXX variable pool other than the current one.

The keyword PRODUCER may be used when the pipeline specification is issued with CALLPIPE. It specifies that the variable pool to be accessed is the one for the stage that produces the input to the stage that issues the subroutine pipeline that contains *rexxvars*, rather than the current stage. (This is a somewhat esoteric option.) To ensure that the variable pool persists as long as this invocation of *rexxvars*, the stage that is connected to the currently selected input stream must be blocked in an OUTPUT pipeline command while the subroutine pipeline is running.

The keyword MAIN specifies that the REXX variable pool to be accessed is the one in effect at the time the pipeline set was created (either by the PIPE command or by the *runpipe* stage). MAIN is implied for pipelines that are issued with ADDPIPE.

A number that is zero or positive is optional. It specifies the number of REXX variable pools to go back. That is, *rexxvars* can operate on variables in the program that issued the pipeline specification to invoke *rexxvars* or in one of its ancestors. (When the number is prefixed by either PRODUCER or MAIN, the variable pool to be accessed is the producer's or the main one, or one of their ancestors.) On CMS, if the number is larger than the number of REXX environments created on the call path from the PIPE command, *rexxvars* continues on the SUBCOM chain starting with the environment active when PIPE was issued.

Specify the option NOMSG233 to suppress message 233 when the REXX environment does not exist. Either way, *rexxvars* terminates with return code 233 on commit level -1 when the environment does not exist.

Specify TOLOAD to write output records in the format required as input to *varset* (and to *varload*): each record contain the variable's name as a delimited string followed by the variable's value. The delimiter is selected from the set of characters that do not occur in the name of the variable; it is unspecified how this delimiter is selected. The keyword COMMENTS is followed by a delimited string that enumerates the characters that should not be used as delimiter characters. The keyword NOCOMMENTS specifies that the delimiter character can be any character that is not in the variable's name. By default, neither asterisk nor blank is used as a delimiter, because these are the default comment characters used by *varload*.

The keyword WIDTH specifies the minimum size in bytes of the buffer into which REXX returns the value of a variable. The number specified is a minimum buffer size; *rexxvars* may allocate more. The default width is 512.

Output Record Format: When TOLOAD is specified, one line is written for each variable in the variable pool. The line contains these items (without additional separators):

1. A delimiter character (in column 1).
2. A variable's name (beginning in column 2).
3. A delimiter character (a copy of the character in column 1).
4. The variable's value.

When TOLOAD is omitted, the first line contains the source string for the REXX environment. Subsequent pairs of records describe variables; a record that contains the name of a variable is followed by a record that contains the variable's value. Each line is prefixed with a character describing the item; three prefix characters are used:

- s The source string. This is the first line written.
- n The name of a variable. The value is on the following line.
- v The data contained in the variable whose name is defined in the preceding record. Only as much data as will fit within the width specified (or 512, the default) are written to the pipeline.

There is one blank between the prefix character and the data.

Commit Level: *rexvars* starts on commit level -1. It verifies that the REXX environment exists (if it did not do so while processing its parameters). It fetches a dummy variable from the pool to ensure that it starts fetching variables at the beginning of the pool and then commits to level 0.

Premature Termination: *rexvars* terminates when it discovers that its output stream is not connected.

See Also: *var*, *vardrop*, *varfetch*, *varset*, and *stem*.

Examples: To dump the current REXX variables to a file for later analysis:

```
/* Sample Syntax error routine */
Syntax:
Say 'Syntax error' RC:' errortext(RC)
parse source . . $$$fn$$$ $$$ft$$$ .
Say 'Error occurred on line' sigl 'of' $$$fn$$$ $$$ft$$$
Say sourceline(sigl)
address command 'PIPE rexvars | >' $$$fn$$$ 'variables a'
```

The instruction Signal on Syntax causes the routine to be invoked whenever there is a syntax error.

To display which REXX program has called a given one:

```
/* Who called me */
'pipe rexvars 1 | take 1 | var caller'
if RC/=0 then exit RC
parse var caller . . . fn ft fm .
parse source . . myfn myft myfm .
say fn ft fm 'called' myfn myft myfm.'
```

The first record written by *rexvars* (in this case the only output record that is used) contains the source string, from which the name of the program can be inferred.

A *buffer* stage is required to buffer the output from *rexvars* when data derived from its output are stored back into the variable pool with a *var*, *stem*, *varload*, or *varset* stage:

```
pipe rexvars | find v_ARRAY. | spec 3-* | buffer | stem vars.
```

As shown in this example, it may be more efficient to buffer the variables that are set rather than the output from *rexvars*.

Notes:

1. *rexvars* uses the EXECCOMM “get next” interface when it processes a REXX variable pool. REXX maintains, with the variable pool, a “cursor” to the next variable it will fetch. The cursor is reset to the beginning when a variable is dropped, fetched, or set, either by the interpreter itself or through the EXECCOMM interface. Thus, if some other stage repeatedly causes the cursor to be reset to the beginning while *rexvars* is

extracting the variables of a pool, an infinite number of records may be written by *rexxvars*.

Clearly, the variable pool will be accessed if the pipeline writes its result back into several variables in the same variable pool (*stem*, *varset*), but there are many other and more subtle variations. For example, no other stage may access the variable pool to fetch variables or drop variables (*var*, *vardrop*, and *varfetch*), lest the cursor be reset.

Thus, the output from a *rexxvars* stage must be buffered (for example, in a *buffer* stage) if anything else in the CMS session could cause REXX to access the variable pool before an unbuffered *rexxvars* stage would terminate. (This includes, but is not limited to, current and future stages in the pipeline set or any pipeline set created by the pipeline.) By inserting a *buffer* stage after *rexxvars*, you allow *rexxvars* to run to completion before subsequent stages can possibly begin manipulating the variable pool.

Special care is needed if the pipeline specification contains stages that access the variable pool, if these stages cannot be proven to be synchronised with the buffered output (that is, if they might access the variable pool before the *buffer* stage produces output), and if there are any stages between *rexxvars* and *buffer*. No stage in this cascade may suspend itself, nor may any stage have secondary streams defined. In this restricted environment the otherwise unspecified pipeline dispatcher will be guaranteed not to run stages outside the pipeline segment up to the *buffer* stage, once it has dispatched *rexxvars* on commit level 0.

2. The output from *rexxvars* is unspecified when more than one *rexxvars* stage accesses a particular variable pool concurrently. As far as REXX is concerned, these stages will be sharing the read cursor and will fetch the variables in the variable pool between them. When REXX comes to the end of the variable pool, it will signal this condition to one of the *rexxvars* stages, which will then terminate. The remaining *rexxvars* stages will then read the variable pool from the beginning. Thus, the *rexxvars* stages will eventually all terminate, but the result is unlikely to be what you were looking for.
3. *rexxvars* obtains variables exposed at the time the pipeline specification is issued. Any variables hidden by a Procedure instruction are not returned by the underlying interface.
4. The underlying interface does not provide the default value assigned to a stem, because it cannot be distinguished from the compound variable with a null index. Note the difference in these assignments:

```
array.=0 /* All existing compounds are reset */
ix=' '
array.ix=1 /* Only one variable is set */
```
5. PIPE var stem.|... reads the default value of a stem.
6. When a pipeline is issued as a TSO command, IKJCT441 is called to access the variable pool. When the command is issued with Address Link or Address Attach, *rexxvars* accesses the REXX environment from where the command is issued. When IKJCT441 is used, the first line written has two words, TSO CLIST, to identify the environment.
7. *CMS Pipelines* maintains a reference to the current variable environment for each stage. Initially this is the environment in effect for the PIPE command with which the original pipeline was started.

When a REXX program is invoked (as a stage or with the REXX pipeline command), its environment becomes the current one, with a pointer to the previous one.

When a pipeline specification is issued with the *runpipe* built-in program or the CALLPIPE pipeline command, the current environment is the one in effect for the stage issuing *runpipe* or CALLPIPE; it is known to persist while the subroutine pipeline runs. On the other hand, when a pipeline specification is issued with the ADDPIPE pipeline command, the stage that issues ADDPIPE runs in parallel with the added pipeline specification; it can terminate at any time (indeed, even before the new pipeline specification starts running). Therefore, for ADDPIPE, the current environment is set to the one for the last *runpipe* or the one at initial entry on the PIPE command. Thus, the MAIN option has effect only for pipeline specifications that are issued by the CALLPIPE pipeline command.

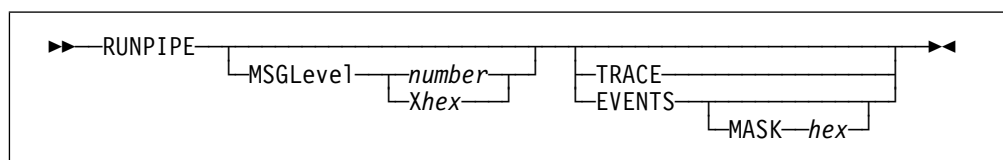
8. *rexxvars* cannot handle truncation of the value of a variable when the buffer is too small, because it cannot retry the call to the underlying interface. Use *varfetch* to ensure you get the complete value:

```
pipe rexxvars|find n|substr 3-*|buffer|varfetch toload|...
```

runpipe—Issue Pipelines, Intercepting Messages

runpipe issues pipeline specifications in the same way that the PIPE command does. *CMS Pipelines* messages issued while the pipeline is running are written to the output stream rather than to the terminal. The TRACE option can be specified to force a trace of all stages running in the new pipeline set.

When the option EVENTS is specified, *runpipe* writes detailed information about the pipeline specification and the progress of its execution. This information is designed to be processed by a program.



Type: Control.

Placement: *runpipe* must not be a first stage.

Syntax Description: The arguments are optional.

MSGLEVEL Specify the message level setting for the pipeline sets that are created by *runpipe*. The value after the keyword can be a decimal number or the letter “x” followed by a hexadecimal string. There must be no blank between the letter and the hexadecimal string. All of sixteen rightmost bits can be set. If MSGLEVEL is omitted, the pipeline sets inherit the message level established by PIPMOD rather than the one active for *runpipe*.

TRACE Force the trace option for all pipeline specifications in the pipeline set. This form of trace cannot be disabled in the individual pipeline specification.

EVENTS Produce event records on the output stream.

runpipe

: MASK Specify a mask for event records to be suppressed. By default, the mask
: is zero, which enables all events records. Mask bit numbering follows
: standard IBM System/360 conventions; the mask for event type 0 is bit
: number 0, which is the leftmost one (X'80000000'). A record is
: suppressed when the corresponding mask bit is one. Beware that the
: hexadecimal number is scanned as a binary number; specify all eight
: hexadecimal digits. If six digits are specified, event records 0 through 7
: cannot be suppressed, because their mask will of necessity be zero. It
: makes no sense to specify four or fewer hexadecimal digits for the mask.

Operation: Input records are issued as pipeline specifications. A new pipeline set is created for each record.

When EVENTS is omitted, the new pipeline runs until it completes or issues a message. When a message is issued, the new pipeline waits while the message is written to the primary output stream of *runpipe*; the new pipeline is resumed when the write completes.

When the keyword TRACE is specified, all pipelines in the new pipeline set are forced to be run with the trace option; this cannot be disabled by options in the individual pipelines. As TRACE produces messages, the trace of the subject pipeline set is also written to the output of *runpipe*.

When EVENTS is specified, records are written by the pipeline specification parser and by the pipeline dispatcher in addition to records for messages issued. The EVENTS option and the MASK option apply to all pipeline specifications in the pipeline set.

The REXX environment for the new pipeline is the one in effect for *runpipe*.

A stall in a pipeline that is issued with *runpipe* does not affect the pipeline that contains the *runpipe* stage. *runpipe* ignores errors when it writes output records, even errors that indicate a stall in the pipeline that contains the *runpipe* stage. That is, error conditions cannot leak between the two pipeline sets. This ensures that the pipeline issued by *runpipe* can terminate in an orderly way, even in the event of severe errors in the controlling pipeline set.

Input Record Format: Each line contains a pipeline specification; the syntax of the line is the same as the syntax of the argument string to PIPE. Specifically, global options are specified in parentheses at the beginning of the line.

Output Record Format: When EVENTS is omitted, the output records contain CMS Pipelines messages issued in response to the pipeline specifications and messages issued with the MESSAGE pipeline command. The complete message is written irrespective of the EMSG setting. The first word (10 or 11 characters) is the message identifier. Programs that process these messages should be able to handle message numbers that have both three and four digits.

When EVENTS is specified, output records are written in the format described in Appendix G, "Format of Output Records from *runpipe* EVENTS" on page 939.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *runpipe* writes all output for an input record before consuming the input record.

See Also: *pipcmd*.

Examples: To present error messages as XEDIT messages when a pipeline is issued during an XEDIT session:

```
/* PIPE XEDIT */
parse arg pipe
address command 'PIPE var pipe | runpipe | xmsg'
r=RC
if r<=0
  then 'emsg Return code:' r'.'
exit r
```

runpipe is useful when tracing the pipeline dispatcher, which often generates large amounts of data:

```
/* debug pipe */
pipe='(trace)' arg(1)
address command
'PIPE var pipe | runpipe | > pipeline trace a'
```

To test if a particular function is available or in general to issue a pipeline and be sure no messages are written to the terminal:

```
/* Now see if it can do it */
'PIPE literal deblock Linend Eof | runpipe'
modern=RC=0
```

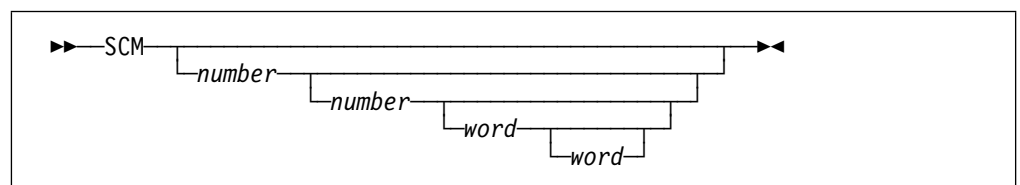
Notes:

1. Do not mask off console input events (type X'0F') as this causes an input *console* stage to produce an infinite number of input lines.

Return Codes: The return code is the aggregate of the return codes from the pipelines that have been run: The aggregate return code is the minimum return code if any return code is negative; otherwise it is the maximum of the return codes received.

scm—Align REXX Comments

scm (“shift comments” or “smarten up comments”) processes REXX and C programs to line up comments and complete unclosed comments. *scm* is designed to be used from an XEDIT or ISPF/PDF macro to format a few lines of a program. *scm* does not support comments that span lines.



Type: Filter.

Syntax Description: The first number specifies the column where a comment should begin when the REXX instruction is short enough to leave room for alignment. The default is 39. The second number specifies the ending column for the comment. The default is 71. The last two words specify the beginning and ending comment strings, respectively.

If both words are omitted, the defaults are `/*` and `*/`. When the first word is specified, the default for the second word is null (ADA-style comments).

Operation: A line is passed to the output unchanged when:

- It has no beginning comment string.
- It has a comment end string that is after the last comment begin string in the line, but not at the end of the line.

When a line has only one comment begin string and this is the first non-blank string on that line, the comment end string is aligned with the ending column (assuming the line does not extend beyond this column).

When the line contains non-blank characters before the last comment string, the string is aligned within the specified columns if the instruction part and the comment are both shorter than the width of their respective column ranges.

When the instruction or the comment is longer than the column range allocated, but together they are shorter than the area allocated, the comment is aligned to the right.

Otherwise the comment is appended to the instruction.

Record Delay: `scm` strictly does not delay the record.

Commit Level: `scm` starts on commit level -1. It verifies its arguments and then commits to 0.

Premature Termination: `scm` terminates when it discovers that its output stream is not connected.

Examples:

```

pipe literal /* Now we check a and b | scm | console
▶ /* Now we check a and b                        */
▶Ready;

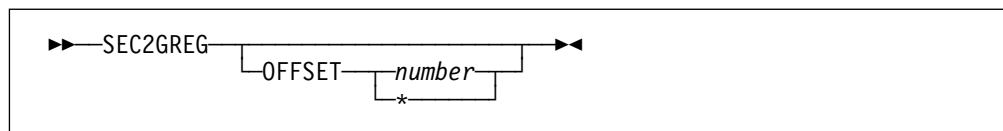
pipe literal If a=b /* The same thing? | scm | console
▶ If a=b                                           /* The same thing?
▶Ready;
```

Notes:

1. Multiline comments (comments that are spanned over several lines) are not supported, because the comment on the first line will be terminated. If such a program is processed by `scm`, it is likely to encounter errors when it is run.
2. `scm` is used in the SCM XEDIT program that is included with *CMS Pipelines*.

sec2greg—Convert Seconds Since Epoch to Gregorian Timestamp

Convert a number representing the seconds since January first, 1970. A negative number represents a point in time before the beginning of the epoch.



Type: Filter.

Syntax Description:

OFFSET A time zone offset is specified.

number Specify a number of seconds. The numerically largest acceptable value is 86399, as a time zone offset of 24 hours makes no sense. Positive values are east of Greenwich.

* Use the time zone offset set for the host system.

Output Record Format: 14 characters without punctuation: *yyyymmddhhmms*.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *sec2greg* does not delay the record.

Premature Termination: *sec2greg* terminates when it discovers that its output stream is not connected.

Converse Operation: *greg2sec*.

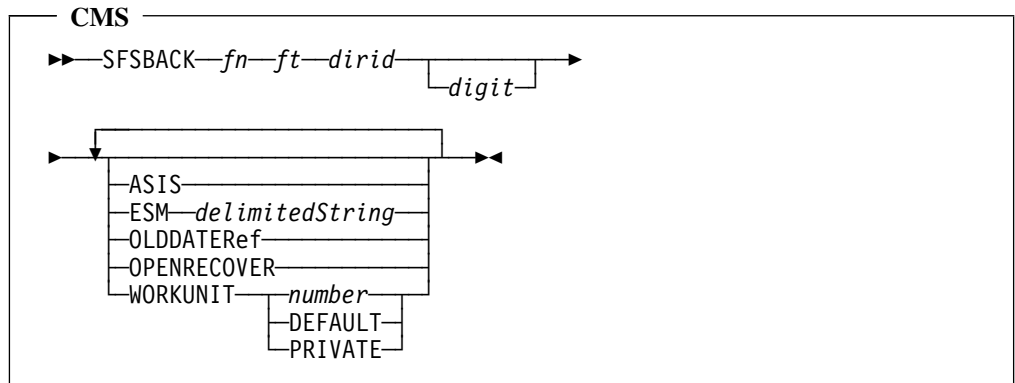
See Also: *dateconvert* and *spec*.

Notes:

1. The epoch started at 00:00:00 UTC on January first, 1970. This is the epoch used in UNIX systems.
2. LOCAL may also be specified to apply the local time zone offset.
3. A time zone offset of 86399 is not the same as one of -1.
4. For dates before year 1970, *sec2greg* ignores all issues as to whether the day actually occurred or the year existed at all.
5. The largest valid input number of seconds is 253402300799, which corresponds to the end of year 9999.
6. Leap seconds are not accounted for, as most UNIX systems also ignore this issue.

sfsback—Read an SFS File Backwards

sfsback reads the last record, then the second last record, and so on. *sfsback* accesses a file that is stored in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode.



Type: Device driver.

Placement: *sfsback* must be a first stage.

Syntax Description:

- fn* Specify the file name for the file.
- ft* Specify the file type for the file.
- dirid* Specify the mode, the directory, or a NAMEDEF for the directory for the file.
- digit* Specify the file mode number for the file.
- ASIS Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified.
- ESM Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
- OLDDATEREF Pass the keyword to the open routine. CMS will not update the date of last reference for the file.
- OPENRECOVER Pass the keyword to the open routine. This particular operation is performed as if the file's attributes were RECOVER and NOTINPLACE.
- WORKUNIT Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is DEFAULT.

Operation: Reading begins at the last record in the file and proceeds backwards to the first record. The file is closed before *sfsback* terminates.

Commit Level: *sfsback* starts on commit level -2000000000. It creates a private unit of work if WORKUNIT PRIVATE is specified, opens the file, allocates a buffer if required, and then commits to level 0.

Premature Termination: *sfsback* terminates when it discovers that its output stream is not connected.

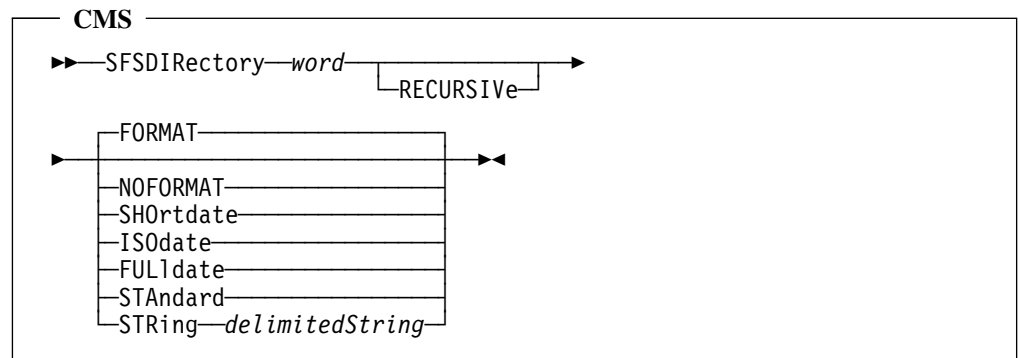
See Also: *disk*, *diskrandom*, *diskslow*, *filetoken*, *members*, and *pdsdirect*.

Examples: To read a file backwards:
 pipe diskback profile exec . | ...

This reads your profile from your root directory in the current file pool. *diskback* selects *sfsback* to process the file, because the third word is present, but does not specify a mode.

sfsdirectory—List Files in an SFS Directory

sfsdirectory writes information about files in the specified directory and, if requested, all subdirectories of the specified directory.



Type: Device driver.

Placement: *sfsdirectory* must be a first stage.

Syntax Description:

<i>word</i>	Specify a mode letter, a name definition, or a directory path. A mode letter must refer to an accessed directory (that is, not a minidisk).
RECURSIVE	List the contents of subdirectories and their subdirectories, and so on.
FORMAT	Information about files is written in a printable format using the short date format.
NOFORMAT	The output record contains the raw format 1 (file) DIRBUFF control block describing the file or subdirectory.
FULLDATE	The file's timestamp is formatted in the American format, with the century: 3/09/1946 23:59:59.
ISODATE	The file's timestamp is formatted with the century in one of the formats approved by the International Standardisation Organisation: 1946-03-09 23:59:59.
SHORTDATE	The file's timestamp is formatted in the American format, without the century: 3/09/46 23:59:59.
STANDARD	The file's timestamp is formatted as a single word in a form that can be used for comparisons: 19460309235959.
STRING	Specify custom timestamp formatting, similar to the POSIX <code>strftime()</code> function. The delimited string specifies formatting as literal text and substitutions are indicated by a percentage symbol (%) followed by a character that defines the substitution. These substitution strings are recognised by <i>sfsdirectory</i> :

sfsrandom

```
:          %%      A single %.  
:          %Y      Four digits year including century (0000-9999).  
:          %y      Two-digit year of century (00-99).  
:          %m      Two-digit month (01-12).  
:          %n      Two-digit month with initial zero changed to blank ( 1-12).  
:          %d      Two-digit day of month (01-31).  
:          %e      Two-digit day of month with initial zero changed to blank ( 1-31).  
:          %H      Hour, 24-hour clock (00-23).  
:          %k      Hour, 24-hour clock first leading zero blank ( 0-23).  
:          %M      Minute (00-59).  
:          %S      Second (00-60).  
:          %F      Equivalent to %Y-%m-%d (the ISO 8601 date format).  
:          %T      Short for %H:%M:%S.  
:          %t      Tens and hundredth of a second (00-99).
```

```
: Operation: A private unit of work is obtained to ensure a consistent view of the file  
: space. A line is written for each file or directory in the specified root directory. When  
: RECURSIVE is specified, the contents of subdirectories are also written.
```

```
: Output Record Format: For NOFORMAT, the output record is 112 bytes. Refer to the  
: macro DIRBUFF in DMSGPI MACLIB for intent FILE and the description of DMSGETDI.
```

```
: Commit Level: sfsdirectory starts on commit level -2000000000. It obtains a unit of  
: work, verifies that the root directory exists, and then commits to level 0.
```

```
: Premature Termination: sfsdirectory terminates when it discovers that its output stream  
: is not connected.
```

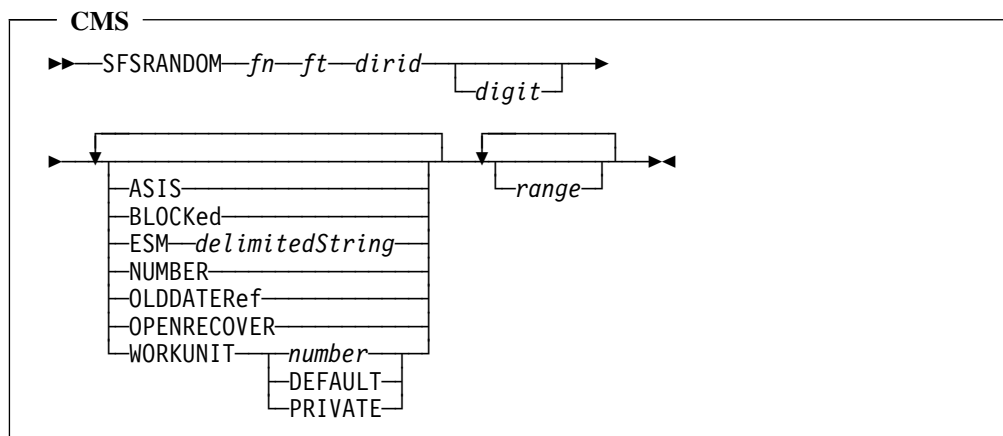
```
: See Also: state and statew.
```

```
: Examples: To list the root directory of the server that runs the samples. It contains a  
: single file:
```

```
:      pipe sfsdir . str /%F/ | console  
:      ▶ARCH                - D                2015-08-13 SFS:P>  
:      ▶DELTA                - D                2015-07-30 SFS:P>  
:      ▶FPLNOTES ENLIGD     -1 V                16384      18      69 2015-09-28 SFS:P>  
:      ▶MASTER              - D                2015-12-04 SFS:P>  
:      ▶PS                   - D                2015-08-04 SFS:P>  
:      ▶Q                    - D                2015-08-13 SFS:P>  
:      ▶UTILS                - D                2015-07-30 SFS:P>  
:      ▶V                    - D                2015-08-13 SFS:P>  
:      ▶Z                    - D                2015-08-13 SFS:P>  
:      ▶TMPHELP             - D                2020-04-29 SFS:P>  
:      ▶Ready;
```

sfsrandom—Random Access an SFS File

sfsrandom reads records in a specified order from a CMS file that is stored in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode.



Type: Device driver.

Syntax Description:

<i>fn</i>	Specify the file name for the file.
<i>ft</i>	Specify the file type for the file.
<i>dirid</i>	Specify the mode, the directory, or a NAMEDEF for the directory for the file.
<i>digit</i>	Specify the file mode number for the file.
ASIS	Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified.
BLOCKED	Write a range of records from the file as a single output record; the file must have fixed record format.
ESM	Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
NUMBER	Prefix the record number to the output record. The field is ten characters wide; it contains the number with leading zeros suppressed.
OLDDATEREF	Pass the keyword to the open routine. CMS will not update the date of last reference for the file.
OPENRECOVER	Pass the keyword to the open routine. This particular operation is performed as if the file's attributes were RECOVER and NOTINPLACE.
WORKUNIT	Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is DEFAULT.

Further arguments are ranges of records to be read. Use an asterisk as the end of a range to read to the end of the file.

Commit Level: *sfsrandom* starts on commit level -2000000000. It creates a private unit of work if WORKUNIT PRIVATE is specified, opens the file, allocates a buffer if required, and then commits to level 0.

Premature Termination: *sfsrandom* terminates when it discovers that its output stream is not connected.

See Also: *disk*, *diskback*, *diskslow*, *filetoken*, *members*, and *pdsdirect*.

Examples: Both of these commands read records 7, 8, 3, and 1 from a file and write them to the pipeline in that order:

```
pipe diskrand profile exec . 7.2 3 1 |...  
pipe literal 3 1 | diskrand profile exec . 7.2 |...
```

This reads your profile from your root directory in the current file pool. *diskrand* selects *sfsrandom* to process the file, because the third word is present, but it does not specify a file mode.

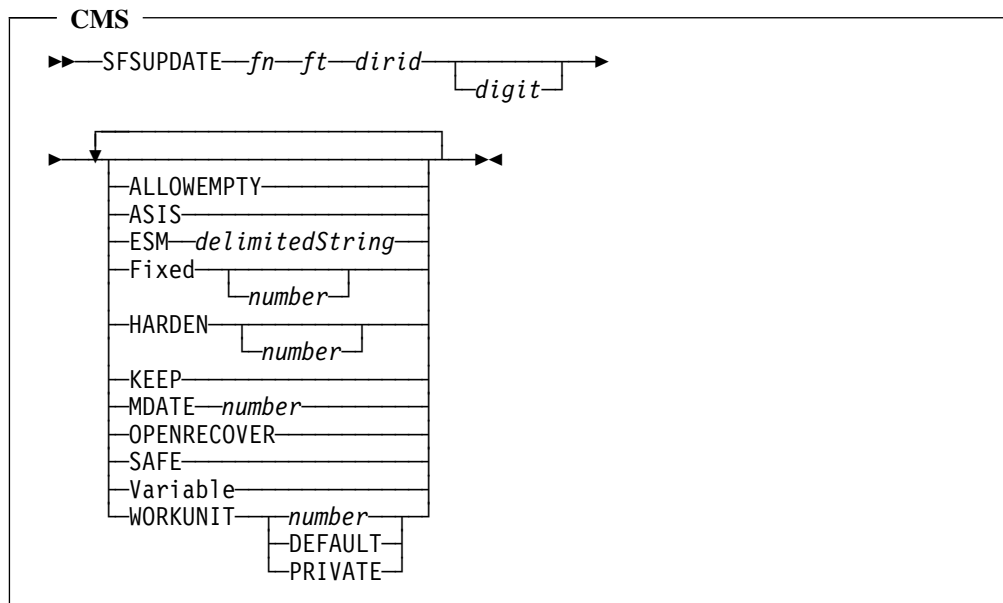
Notes:

1. RECNO is a synonym for NUMBER.
2. *sfsrandom* performs at least one read operation for the records in the arguments, if specified, and one read operation for each input record. When BLOCKED is specified, all records in a range are read in a single operation. It is unspecified how many additional read operations it performs for records specified in the arguments or a particular input record. This may be significant when the file is updated with *diskupdate*. Ensure that no stage delays the record between stages reading and writing a file being updated.
3. When no options are specified and the first *range* consists of a single *number* less than 7, the file mode number is required to avoid the first range being interpreted as file mode number. Since the file mode number is ignored when reading an SFS file, any valid number may be specified. To avoid confusion, provide the ranges through input records to *sfsrandom*.

!
!
!
!
!

sfsupdate—Replace Records in an SFS File

sfsupdate replaces records in or appends records to a file in the Shared File System (SFS) directly, using a directory path or a NAMEDEF. The directory need not be accessed as a mode. A file is created if one does not exist.



Type: Device driver.

Placement: *sfsupdate* must not be a first stage.

Syntax Description:

- fn* Specify the file name for the file.
- ft* Specify the file type for the file.
- dirid* Specify the mode, the directory, or a NAMEDEF for the directory for the file.
- digit* Specify the file mode number for the file.
- ALLOWEMPTY Pass the keyword to the open routine. When ALLOWEMPTY is specified, CMS creates an empty file if no input is read. The default is not to create a file when there is no input.
- ASIS Use the file name and file type exactly as specified. The default is to translate the file name and file type to upper case when the file does not exist as specified.
- ESM Provide a character string for an external security manager. The character string can be up to eighty characters and it may contain blanks.
- FIXED The record length may be specified after FIXED. Create a new file with fixed record format; verify that an existing file has fixed record format. If the record length is specified and the file exists, it is verified that the file is of the specified record length.
- HARDEN Perform SFS commit operations to make the file contents permanent before end-of-file is read. Specify the number of records to write to the file between each SFS commit operation. HARDEN is mutually exclusive with SAFE.

!

sfsupdate

!	KEEP	KEEP is ignored unless WORKUNIT PRIVATE is specified or defaulted. When KEEP is specified, changes are committed to the file even when an error has occurred. The default is to roll back the unit of work. KEEP is mutually exclusive with SAFE.
	MDATE	Specify the file modification date and time. The timestamp contains eight to fourteen digits. The first eight digits specify the year (four digits), the month (two digits), and the day (two digits). The remaining digits are padded on the right with zeros to form six digits time consisting of the hour, the minute, and the second. A twenty-four hour clock is used.
	OPENRECOVER	Pass the keyword to the open routine. This particular operation is performed as if the file's attributes were RECOVER and NOTINPLACE.
!	SAFE	SAFE is rejected if WORKUNIT PRIVATE is neither specified nor defaulted. When SAFE is specified, <i>sfsupdate</i> performs a pipeline commit to level 1 before it returns the unit of work. It rolls back the unit of work if the commit does not complete with return code 0. SAFE is mutually exclusive with HARDEN and KEEP.
	VARIABLE	The record length may be specified after VARIABLE. Create a new variable record format file; verify that an existing file has variable record format.
	WORKUNIT	Specify the work unit to be used. You can specify the number of a work unit you have allocated by the DMSGETWU callable service; you can specify DEFAULT, which uses the default unit of work; or you can specify PRIVATE, which gets and returns a work unit for the stage's exclusive use. The default is PRIVATE.

Operation: Columns 11 through the end of the input record replace the contents of the record in the file. The file is closed before *sfsupdate* terminates.

Input Record Format: The first 10 columns of an input record contain the number of the record to replace in the file (the first record has number 1). The number does not need to be aligned in the field. It is an error if an input record is shorter than 11 bytes.

The valid values for the record number depends on the record format of the file:

Fixed For fixed record format files, any number can be specified for the record number (CMS creates a sparse file if required). An input record can contain any number of consecutive logical records as a block. The block has a single 10-byte prefix containing the record number of the first logical record in the block.

Variable When the file has variable record format, the record number must be at most one larger than the number of records in the file at the time the record is written to it. The data part of input records must have the same length as the records they replace in the file.

Streams Used: *sfsupdate* copies the input record (including the record number) to the output after the file is updated with the record.

Record Delay: *sfsupdate* strictly does not delay the record.

Commit Level: *sfsupdate* starts on commit level -2000000000. It creates a private unit of work if `WORKUNIT PRIVATE` is specified or defaulted, opens the file, allocates a buffer if required, and then commits to level 0.

See Also: `>`, `>>`, *disk*, *diskslow*, and *filetoken*.

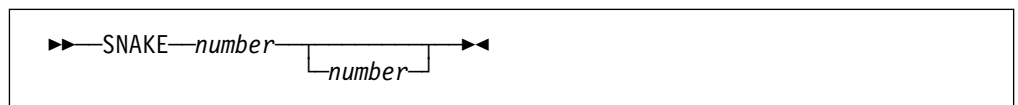
Examples: To replace records in a file with a particular key:

```
/* Update file */
'PIPE (end ?)',
'| < input file .',
'| spec number 1 1-* next',
'|c: change casei 11-* /apples/banana/',
'| sfsupdate input file .',
'?c:'
```

Note that *sfsupdate* runs on a private unit of work, whereas `<` runs on the default unit of work. Defining the secondary output stream to *change* makes it write only changed records to its primary output stream.

snake—Build Multicolumn Page Layout

snake breaks the input file into columns of the specified depth and pastes the columns together side by side. Thus, the input file wiggles its way across the page like a snake.



Type: Filter.

Syntax Description: The first number specifies the number of columns to be made. The second number specifies the number of lines on a “page”. If the second number is omitted, *snake* reads the file and determines the minimum number of rows required to fill all columns; when the number of input records is not evenly divisible by the number of columns, the last column will not be filled completely.

Operation: When the second number is omitted, *snake* reads the entire file to determine the number of records and sets the page depth accordingly.

Assuming the number of lines on a page is n , the first output line contains records 1, $n+1$, $2*n+1$, and so on. Thus, if the input records are sorted, the columns on the page will be sorted downwards.

Input Record Format: Input records should be of fixed length; *snake* neither pads nor truncates to fit records into columns.

Record Delay: *snake* can delay all records that make up a “page”.

Commit Level: *snake* starts on commit level -1. It verifies its arguments and then commits to 0.

Premature Termination: *snake* terminates when it discovers that its output stream is not connected.

See Also: *join*.

socka2ip

Examples: To arrange letters in a square:

```
pipe literal a b c d | split | snake 2 | console
▶ac
▶bd
▶Ready;
```

To transpose a matrix:

```
pipe literal c d | literal a b | cons | split | snake 2 | console
▶a b
▶c d
▶ac
▶bd
▶Ready;
```

The first two lines of output show the input matrix; the last two show the resulting matrix without padding.

socka2ip—Format `sockaddr_in` Structure

socka2ip converts input records that are sixteen bytes to a readable port number and IP address; it converts input records that are four bytes to a readable IP address.

▶▶—SOCKA2IP—◀◀

Type: Filter.

Input Record Format: When the input line is four bytes long, the input record contains the unsigned long IP address to be converted.

Otherwise the input record contains a structure of sixteen bytes. Binary numbers are stored in the network byte order, that is, with the most significant bit leftmost.

Pos	Len	Description
1	2	The short unsigned number 2, which specifies that the addressing family is AF_INET.
3	2	The short unsigned port number.
5	4	The unsigned IP address.
9	8	Reserved. Binary zeros

Output Record Format: When the input line is four bytes long, the output record contains a single word, which is the IP address in dotted-decimal notation.

Otherwise the output record contains three blank-delimited words:

1. The literal constant “AF_INET”.
2. The port number, which is in the range 0 to 65535, inclusive.
3. The IP address in dotted-decimal notation.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *socka2ip* does not delay the record.

Premature Termination: *socka2ip* terminates when it discovers that its output stream is not connected.

Converse Operation: *ip2socka*.

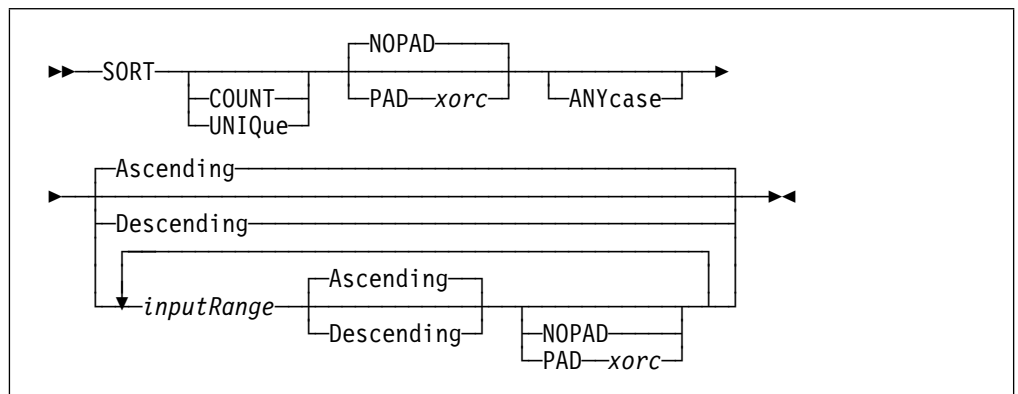
Examples:

To format a socket address structure:

```
pipe strliteral x0002001101020304 | pad 16 00 | socka2ip | console
▶AF_INET 17 1.2.3.4
▶Ready;
```

sort—Order Records

sort reads all input records and then writes them in a specified order.



Type: Sorter.

Syntax Description: Arguments are optional. If present, the keywords COUNT or UNIQUE must be first. Write the keywords PAD or NOPAD in front of the sort fields to specify the default for all fields; the default is NOPAD. The keyword NOPAD specifies that key fields that are partially present must have the same length to be considered equal; this is the default. The keyword PAD specifies a pad character that is used to extend the shorter of two key fields.

The keyword ANYCASE specifies that case is to be ignored when comparing fields; the default is to respect case. Up to 10 sort ranges can be specified. The default is to sort ascending on the complete record. The ordering can be specified for each field; it is ascending by default.

Operation: Records with identical sort keys remain in the order they appear on input unless one of the keywords COUNT or UNIQUE is used.

The first record with a given key is retained when COUNT or UNIQUE is used; subsequent records with duplicate keys are discarded. A 10-character count of the number of occurrences of the key is prefixed to the output record when COUNT is specified.

Streams Used: Records are read from the primary input stream and written to the primary output stream. *sort* reads all input records before it writes output. When COUNT or UNIQUE is specified, records that have duplicate keys are written to the secondary output

stream, if it is defined and connected. End-of-file on the secondary output stream is ignored. When COUNT and UNIQUE are omitted, only one stream may be defined.

Record Delay: *sort* delays all records until end-of-file.

Commit Level: *sort* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

See Also: *collate*, *dfsort*, *lookup*, *merge*, and *unique*.

Examples: To sort hexadecimal data correctly, use *xlate* to change the collating sequence so that A through F sort after the numbers. Use another *xlate* to change the sequence back after the sort. It is assumed that the sort field contains only blanks, numbers, and A through F.

```
/* HEXSORT REXX: Sort in HEX */
'callpipe',
  '*: |',
  'xlate *-* A-F fa-ff fa-ff A-F |',
  'sort' arg(1) '|',
  'xlate *-* A-F fa-ff fa-ff A-F |',
  '*:'
exit RC
```

sort sorts binary data, even when the data may look like numbers, which you might expect to be sorted numerically rather than by the collating sequence:

```
pipe literal 11 5 2 1 | split | sort | console
▶1
▶11
▶2
▶5
▶Ready;
```

See “Numeric Sorting” on page 128 for an approach to sort this like numbers.

Notes:

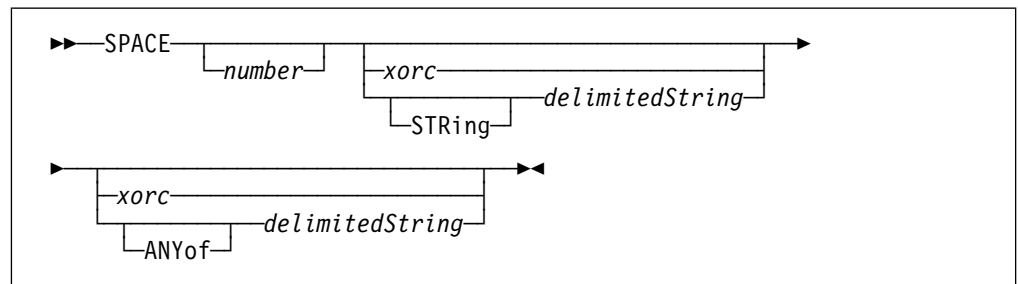
1. *sort* is stable. That is, records that have the same contents of the key field(s) are in the same order on output as they were on input.
2. Use DFSORT/CMS, IBM Program Number 5664-325, to sort files that are too large for *sort*. *dfsort* can be used to interface *CMS Pipelines* to this sort program.
3. Unless ANYCASE is specified, key fields are compared as character data using the IBM System/360 collating sequence.
4. Use *spec* (or a REXX program) for example to put a sort key in front of the record if you wish, for instance, to use a numeric field that is not aligned to the right within a column range. Such a temporary sort key can be removed with *substr* for example after the records are written by *sort*.
5. Use *xlate* to change the collating sequence of the file.
6. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
7. Note in particular that *sort* performs a binary comparison of key fields from left to right. Thus, numeric fields will be sorted “correctly” only when the data to be compared are aligned to the right within sort fields of equal size. (Since padding is applied on the right hand side only.) Thus, a numeric sort is unlikely to “work” when

the sort field is defined as, for example, a word. See also “Numeric Sorting” on page 128.

8. *sort* UNIQUE orders the file and discards records with duplicate keys. Refer to *lookup* for an example of extracting all unique records from a file without altering their order.

space—Space Words Like REXX

space is a generalisation of the REXX *space* built-in function.



Type: Filter.

Syntax Description:

- | | |
|---------------|--|
| <i>number</i> | Specify the number of occurrences of the pad string to insert for each internal string of delimiters. The default is 1. |
| STRING | The second operand specifies the string to replace sequences of delimiter characters. The default is a single blank. |
| ANYOF | The third operand specifies the delimiter character(s). This string contains an enumeration of characters that are all considered to be delimiters. The default is a single blank. |

Operation: Leading and trailing delimiters are removed from the record. Internal sequences of delimiter characters are replaced by the specified number of the replacement string.

Record Delay: *space* strictly does not delay the record.

Premature Termination: *space* terminates when it discovers that its output stream is not connected.

Examples:

space

```
. pipe literal a b b |space|insert /*/ after | console  
. ▶a b b*  
. ▶Ready;
```

```
. pipe literal a b b |space blank|insert /*/ after | console  
. ▶a b b*  
. ▶Ready;
```

```
. pipe literal 3.6=7,3 |space 0 /,./ | console  
. ▶3673  
. ▶Ready;
```

```
. pipe literal a b c d |space /*/ | console  
. ▶a(*)b(*)c(*)d  
. ▶Ready;
```

```
. pipe literal a b c d |space 2 /*/ | console  
. ▶a(*) (*)b(*) (*)c(*) (*)d  
. ▶Ready;
```

Notes:

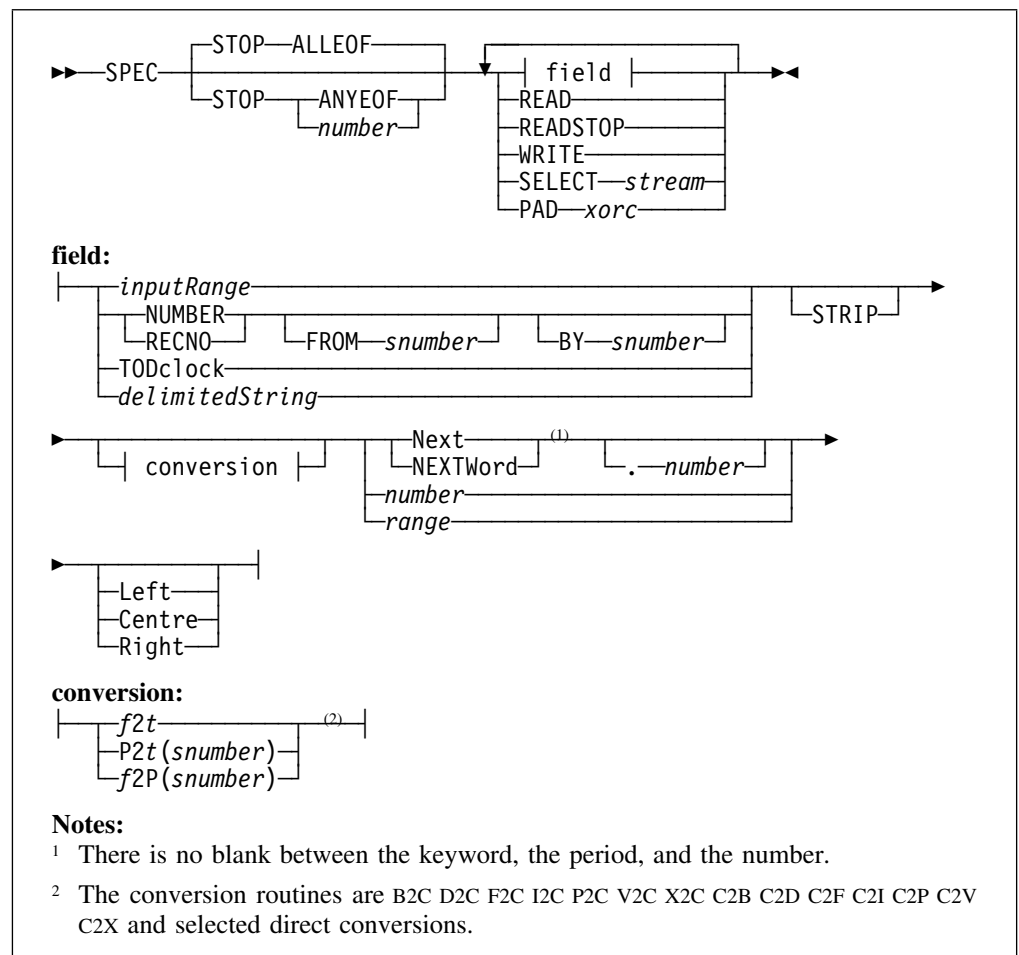
1. When specifying a numeric replacement string of one or two characters (which parses as an *xorc*), you must specify a count even when you wish the default of 1.
2. When just one string argument is specified, it is taken to be the replacement string unless ANYOF is specified or the count is zero.
3. Unlike the REXX built-in function, *space* supports replacement strings longer than one character and more than one blank character.

spec—Rearrange Contents of Records

spec builds output records from the contents of input records and literal fields. It does this by processing a list of specifications (a *specification list*) for each input record. *spec* was originally written to perform the function of the SPECS option on the CMS COPYFILE command, but it has long since expanded far beyond moving columns to rearrange a record. Though the syntax diagram below may look intimidating, *spec* is governed by a few simple concepts that you will find easy enough to learn.

spec can convert the contents of fields in several ways. It can generate an output record containing data from several input records, and it can generate several output records from a single input record.

This article contains but an overview of *spec*. Refer to Chapter 16, “*spec* Tutorial” on page 166 and Chapter 24, “*spec* Reference” on page 719.



Type: Filter.

Syntax Description: Specify `STOP` to terminate *spec* when it discovers that a specified number of input streams are at end-of-file. The default is to process all input streams to end-of-file.

Specify a list of one or more items. Each item defines a field in the output record or contains a keyword to control processing.

The specification of a field in the output record consists of:

- The source of the data to be loaded. It can refer to the input record; it can refer to symbolic fields maintained by *spec*; or it can be a literal argument. When the input record is the source of the data, the extent of the input field can be specified as a column range, a range of blank-delimited words, or a range of tab-delimited fields. The input field can also consist of a single column (or word or field); its position can be relative to the beginning or the end of the input record.
- An optional keyword to specify that the input field is to be stripped of leading and trailing blanks before further processing.
- An optional conversion routine. A conversion routine is specified as three characters that has the digit 2 (an abbreviation for the word “to”) in the middle. The first and the last character must both be one of the characters BCDFIPVX. The first letter specifies the format of the input field; the last letter specifies the format to which the field is to be converted. Not all combinations are supported (some make no sense). When one of the formats is P (for packed), the name of the conversion routine may be followed immediately by parentheses containing a signed number.
- The position of the field in the output record. This can be a column number, a range, or the keywords NEXT or NEXTWORD.
- An optional keyword to specify the placement of the data within the output field. This can be LEFT, CENTRE, or RIGHT.

These keywords specify functions of *spec* that are not related to formatting output fields:

READ	Read another record from the currently active stream (after discarding the current record). A stream at end-of-file is considered to contain a null record.
READSTOP	Perform like READ, but terminate the pass over the specification list if the stream is at end-of-file.
WRITE	Write an output record containing the data from the specification items processed so far in the list.
SELECT	Switch to another input stream and process the record available on that input stream. The word following SELECT should be a stream number or a stream identifier of the input stream to which the following specification items refer.
PAD	Use the specified character to pad short fields when storing subsequent items in the output record. The word following PAD specifies the character to be used as the pad character; it can be specified as a single character, a two-character hexadecimal code, or as one of the keywords BLANK or SPACE.

Operation: Output records are built from literal data and fields in input records, which can come from multiple streams. The output record is built in the order the items are specified. The length of a literal field is given by the length of the literal itself; TODCLOCK is eight bytes long; NUMBER is 10 bytes long. A copied input field extends to the end of the input record or the ending column of the input field, whichever occurs first.

The output record is at least long enough to contain all literal fields and all output fields defined with a range. It is longer if there is an input data field beyond the last literal field, and the input record does contain at least part of the input field.

Padding: Pad characters (blank by default) are filled in positions not loaded with characters from a literal or an input field. The keyword PAD sets the pad character to be used when processing subsequent specification items.

Input field: An input range specification defines a substring of an input record. Depending on the length of a record, an input range may be present in full, partially, or not at all. Input ranges not present in a record are considered to be null; that is, of length zero.

The beginning and end of an input range are, in general, defined by a pair of numbers separated by a semicolon (for example, 5;8). An unsigned number is relative to the beginning of the record; a negative number is relative to the end of the record. None, any one, or both of the numbers may be negative. When the two numbers have the same sign, the first number must be less than or equal to the second number.

When both numbers in the range are unsigned, a hyphen may be used as the separator rather than a semicolon. A range relative to the beginning of a record may also be specified as two numbers separated by a period, denoting the beginning of the range and its length, respectively.

An input range with no further qualification denotes a range of columns. WORDS may be prefixed to indicate a word range; FIELDS may be prefixed to indicate a field range.

The record number: You can put the number of each record into the record, for instance, to generate a sequence field. The keyword NUMBER (with synonym RECNO) describes a 10-byte input field generated internally; it contains the number of the current record, right aligned with leading blanks (no leading zeros). Records are numbered from 1 (the numeral, one) with the increment 1 (the numeral, one) when no further keywords are specified. The word after the keyword FROM specifies the number for the first record; it can be negative. The word after the keyword BY specifies the increment; it too can be negative. The keywords apply to a particular instance of NUMBER. When the record number is negative, a leading minus sign is inserted in front of the most significant digit in the record number, unless the number has ten significant digits (in which case there is no room for the sign). In applications where the 10-digit format is a concern, a counter can be incremented as required and printed in the required format.

The Time-of-day Clock: The contents of the time-of-day clock are stored when a set of input records is ready to be processed. The field is a 64-bit binary counter. It is constant while output record(s) are built. Refer to *Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201, for a description of the time-of-day clock.

Literal field: This is a constant that appears in all output records. A literal character string is written as:

- A delimited string (*delimitedString*) consisting of a character string between two occurrences of a delimiter character, which cannot occur in the string. The delimiter character cannot be blank. It is suggested that a special character be used for the delimiter, but this is not enforced. However, when an alphanumeric character is used as the delimiter, there is a possibility that today's delimited string might become tomorrow's keyword.
- A hexadecimal literal consisting of a leading "x" or "h" (in lower case or upper case) followed by an even number of *hex* characters.
- A binary literal consisting of a leading "b" (in lower case or upper case) followed by zero and one characters in multiples of eight.

Stripping: The keyword STRIP specifies that the field (input field, sequence number, time of day, or literal) is to be stripped of leading and trailing blanks before conversion (if any) and before the default output field size is determined.

Conversion: A field (input or literal) is put in the output record as it is when no conversion is requested for the item. Put the name of a conversion routine between the input and output specifications when you wish to change the format of a field. The functions also defined for REXX work in a similar way. They are C2D, D2C, C2X, and X2C. The functions not available in REXX convert bit strings, floating point, dates, packed decimal, and varying length strings. Note that the REXX name for a conversion function can be misleading: For instance, C2D is described in the REXX manual as converting from character to decimal; what it does, however, is convert from the internal IBM System/390* two's complement notation of a binary number to the external representation in base-10, zoned decimal.

Some conversions are supported directly between printable formats, for example X2B. This table summarises the supported combinations. A plus indicates that the combination is supported. A blank indicates that the combination is not supported.

CDXB	FVPI	To
+++	++++	From C
+	++	From D
++	+	From X
+++	++++	From B
+	++	From F
+	++	From V
+	++	From P
+	++	From I
CDXB	FVPI	To

Composite conversion (x2y) is performed strictly via the C format; that is, x2C followed by C2y.

Output field position: The output specification can consist of the keywords NEXT or NEXTWORD, a column number, or a column range. NEXT indicates that the item is put immediately after the rightmost item that has been put in the output buffer so far. NEXTWORD appends a blank to a buffer that is not empty before appending the item. (A field placed with NEXT or NEXTWORD can be overlaid by a subsequent specification indicating a specific output column.) Append a period and a number to specify an explicit field length with the keywords NEXT and NEXTWORD.

Output field length: Fields for which an explicit length is specified are always present in the output record. Input fields that are not present in the input record or have become null after stripping caused by the STRIP keyword are not stored in the output record. A null literal field is stored in the output record. The default length of the output field is the length of the input field **after** conversion (but **before** placement).

Placement of data in the output field: When an output range is specified without a placement option, the input field after conversion is aligned on the left (possibly with leading blank characters), truncated or padded on the right with pad characters.

A placement keyword (LEFT, CENTRE, CENTER, or RIGHT) is optional in the output field definition. If a placement option is specified, the input field **after** conversion (and thus **after** the length of the output field is determined) is stripped of leading and trailing blank characters unless the conversion is D2C, F2C, I2C, P2C, or V2C.

LEFT The field is aligned on the left of the output field truncated or padded on the right with pad characters.

CENTRE The field is loaded centred in the output field truncated or padded on both sides with pad characters. If the field is not padded equally on both sides, the right side gets one more pad character than the left side. If the field is not

truncated equally on both sides, the left side loses one more character than the right side.

RIGHT The field is aligned to the right in the output field, truncated on the left or padded on the left with pad characters.

Multiple records: You can write input fields from consecutive input records to an output record. Use the keywords **READ** and **READSTOP** to consume a record and peek (read without consuming) the next record on the stream specified by the most recent **SELECT** (or the primary stream if there is no prior **SELECT**). When **READ** is used, a null record is assumed if the stream is at end-of-file. When **READSTOP** is used, end-of-file causes *spec* to write the output record built so far and terminate processing of that set of input records.

READ is convenient, for example, to process the primary stream from *lookup* when it has both master and detail records. (Do not use the **READ** keyword if you wish to write one output record for each input record; a read on all used streams is implied at the end of the specification list.)

You can write multiple output records based on the contents of an input record (or a set of input records). The keyword **WRITE** writes the output record built so far to the primary output stream, leaving the current output record empty.

Streams Used: The keyword **SELECT** specifies that subsequent input fields refer to the specified input stream, which is specified by number or stream identifier. **SELECT 0** is implied at the beginning of the specification list unless an explicit selection occurs before the first input specification referring to a field in an input record. When more than one input stream is selected, a record is peeked (read in locate mode) from all specified input streams before the list is processed. An input stream at end-of-file is considered to hold a null record. Unless **READ**, **READSTOP**, or **WRITE** are used to read or write during the cycle, a set of input records is consumed (released with a move mode read) after the output record is written at the end of the cycle, before further input is obtained.

Input streams defined, but not referenced, are not read when **SELECT** is used. Only the primary input stream is used when the specification list has no **SELECT** keyword (all other streams are ignored). When **STOP ALLEOF** is specified (this is the default), *spec* processes input records until all input streams being used are at end-of-file. When **STOP ANYEOF** is specified, *spec* terminates when it encounters the first stream at end-of-file. When a number is specified, *spec* terminates when that number of streams are at end-of-file, or when all used streams are at end-of-file. The test for termination is performed only when *spec* is reading input records at the beginning of the specification list. End-of-file on a **READ** item does not terminate *spec* immediately; end-of-file on a **READSTOP** causes *spec* to write the output record and terminate processing of the current set of input records.

Record Delay: *spec* synchronises the referenced input streams. It does not delay the record, unless **READ** or **READSTOP** is used.

Commit Level: *spec* starts on commit level -2. It verifies that the primary output stream is the only connected output stream, processes the arguments, and then commits to level 0.

Premature Termination: *spec* terminates when it discovers that any of its output streams is not connected.

See Also: *change*, *chop*, *insert*, *overlay*, and *timestamp*.

Examples: To append an asterisk to each line on the primary input stream:

spec

```
... | spec 1-* 1 /*/ next | ...
```

To generate SEQ8 sequence fields (the record number in columns 73-76 and 77-80 zero):

```
... | spec 1.72 1.72 pad 0 number 73.4 right ?0000? 77 |...
```

Columns 1 to 72 are copied across in the first specification item; the output field size ensures that short records are padded with blanks up to 72 characters. The pad character is then set to 0 so that the leading blanks in the record number are stored as leading zeros; the four rightmost characters of the record number are put in columns 73 to 76 and four zeros appended to make the output record 80 bytes.

To prefix each record (assuming it is shorter than 64K) with a fullword that contains the length of the data part of the record:

```
... | spec x0000 1 1-* v2c next |...
```

A literal with two bytes of binary zeros is put in front of the halfword length generated by the conversion.

To prefix a record with a length field that is two plus the length of the record, as done in structured fields:

```
... | spec 1-* 1 /xx/ next | spec 1-* v2c 1 | spec 1;-3 1 |...
```

This example uses three *spec* stages. The first one appends two characters to the record (it does not matter what these two characters are); the second generates a halfword length field counting these two characters; and the last one removes the two characters, leaving the original record with the required length field in front.

To obtain the contents of the first structured field in the record (the converse of the previous example):

```
... | spec 1-* 1 /xx/ next | spec 1-* c2v | spec 1;-3 | ...
```

To number records starting from zero with leading blanks suppressed:

```
... | spec number from 0 strip 1 1-* nextword | ...
```

Specify a scaling of zero to append a decimal point to the number being unpacked:

```
pipe strliteral x123c | spec 1-* c2p | console
▶+123
▶Ready;
pipe strliteral x123c | spec 1-* c2p(0) | console
▶+123.
▶Ready;
pipe strliteral x123c | spec 1-* c2p(1) | console
▶+12.3
▶Ready;
```

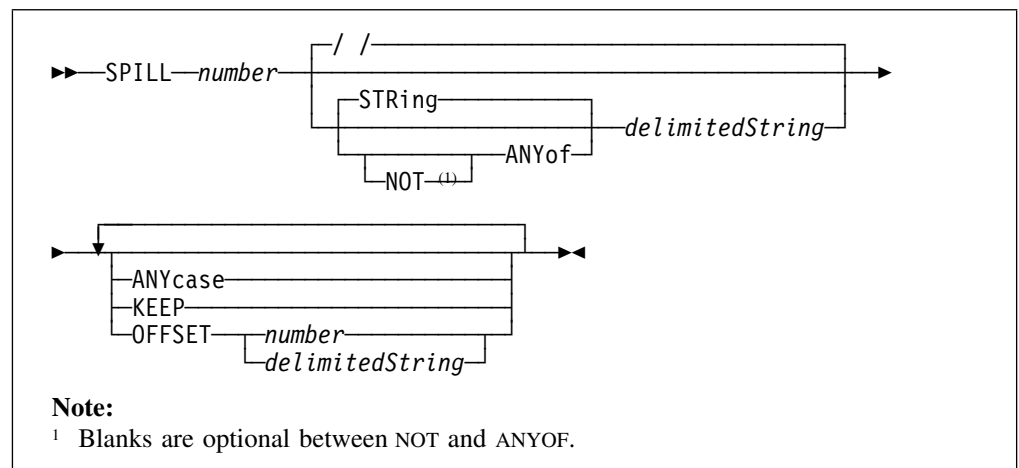
Notes:

1. Some simple functions of *spec* can be done easier with special purpose programs like *substr* and *insert*.
2. In early releases of *CMS Pipelines* *spec* was called *specs*. Gradually the association with COPYFILE has been lost; most users of *CMS Pipelines* refer to the built-in as *spec*. For compatibility reasons, *specs* is retained as a synonym.

3. Floating point conversion (F2C and C2F) requires extended precision floating point hardware.
4. Conversion to floating (F2C) is in most cases accurate within rounding of the least significant bit.
5. C2F conversion can show the effect of rounding errors in the least significant digit when the exponent is close to the limits of the representation.
6. When the specification is a single field with no output column and without PAD, SELECT, READ, or WRITE, the output placement is assumed to be column 1.
7. Unlike the keywords FIELDSEPARATOR, PAD, and SELECT apply to the remainder of the item list; FIELD and WORDS apply to only one input field.
8. An asterisk is rejected for the ending column in an output specification.
9. The time-of-day clock is stored by the machine instruction STCK. In general, this is the time at the primary meridian. A local time zone offset is not applied.
10. RECNO is a synonym for NUMBER. CENTER is a synonym for CENTRE. The keyword NWORD is a synonym for NEXTWORD; it can be abbreviated to two characters. The keyword FS is a synonym for FIELDSEPARATOR. The keyword WS is a synonym for WORDSEPARATOR.

spill—Spill Long Lines at Word Boundaries

spill splits lines longer than a specified number into multiple output lines. Unlike *deblock* FIXED, *spill* splits at word boundaries.



Type: Filter.

Syntax Description: A positive number is required as the first operand. The second positional operand specifies the word separator; it is optional. Remaining operands are optional and may be specified in any order.

number Specify the maximum output record length.

STRING The word separator is a string of characters. If the delimited string contains more than one character, the word separator consists of the specified characters in the order shown.

spill

ANYOF	Any one of the characters enumerated in the delimited string is a word separator.
NOT ANYOF	Any one of the characters not enumerated in the delimited string is a word separator. (This is the complement set.)
ANYCASE	Ignore case when comparing for the word separator. The default is to respect case.
KEEP	Retain the word separator in the output record. The default is to strip word separators at the split point.
OFFSET	Specify the indent on the second and subsequent output lines for an input record. A number specifies the number of blanks to insert; a delimited string specifies the actual string to insert. Output records are not offset when the number is zero or the string is null; this is the default.

Operation: Input records that are shorter than the specified length are passed unchanged to the output.

A leading string is split off long input records until the remainder is not longer than the specified length. The remainder is then passed unmodified to the output with offset applied.

For the first output record for a long input record, the split position is at the specified length or before; for subsequent records, the split point is at the specified length less the length of the offset. The split point is established at the rightmost occurrence of the word separator within the specified range, or abutting the range on the right.

If no word separator can be found within the required range, further processing depends on whether the secondary output stream is defined or not. When the secondary output stream is not defined, the split point is then established within a word. When the secondary output stream is defined, no further attempts are made to split the record. Instead, the remainder of the input record is written to the secondary output stream. It is prefixed with the offset if one or more records were written to the primary output stream before the long word was encountered.

Unless KEEP is specified, word separators are discarded when records are split; this can lead to complete record segments being discarded on the output.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *spill* does not delay the last record written for an input record.

Commit Level: *spill* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *spill* terminates when it discovers that any of its output streams is not connected.

Converse Operation: *join*.

See Also: *chop*, *deblock*, and *split*.

Examples: To flow a paragraph:

```
pipe literal This is a paragraph to be flowed.| spill 12 | console
▶This is a
▶paragraph to
▶be flowed.
▶Ready;
```

To flow an item in a numbered list:

```
pipe literal 2. A most important item.| spill 14 offset 4 | console
▶2. A most
▶ important
▶ item.
▶Ready;
```

When there is no secondary output stream, very long words are split, but not discarded:

```
pipe literal abcdefghi klmnopq| spill 4 | console
▶abcd
▶efgh
▶i
▶klmn
▶opq
▶Ready;
```

When the secondary output stream is defined, a very long word results in the rest of the record being written to that stream:

```
pipe (end ?) literal abc defghi klmn opq| s: spill 4 | hole ? s: | ...
... console
▶defghi klmn opq
▶Ready;
```

```
pipe (end ?) literal abc defghi klmn | s: spill 4 offset 2 | ...
... hole ? s: | console
▶ defghi klmn
▶Ready;
```

In the example above, the primary output stream from *spill* is discarded. When the second word is processed, it is determined that the word cannot be split within the columns allowed and the remainder of the record is written to the secondary output stream.

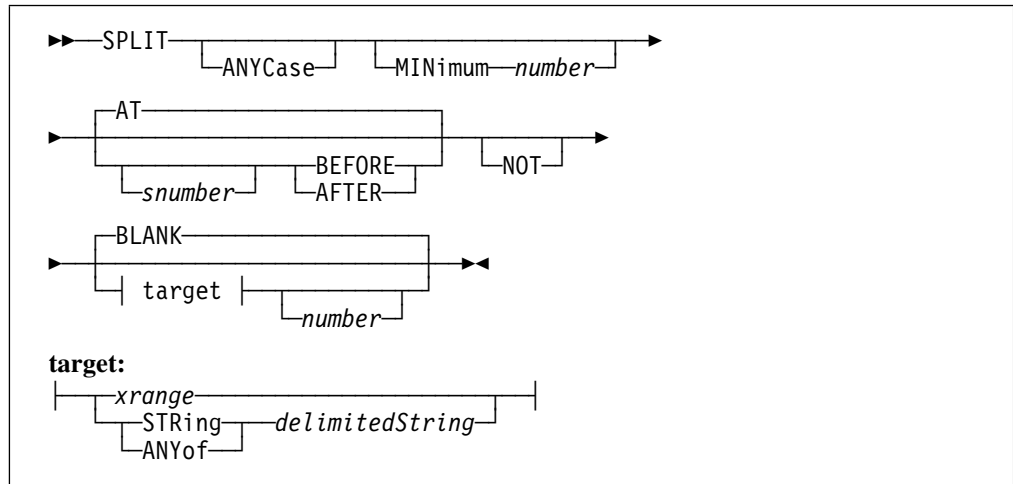
Notes:

1. *spill* is designed to perform a function similar to XEDIT's SET SPILL WORD; though it has several enhancements, it is not suitable as a word processor.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

split

split—Split Records Relative to a Target

split writes one or more records based on the contents of an input record; each part ends before or after a specified character or string.



Type: Filter.

Syntax Description: No arguments are required.

ANYCASE Ignore case. Conceptually, all processing is done in upper case.

MINIMUM Specify a positive number. *split* will not split within the specified number of bytes from the beginning of the record or from a split position.

A relative position and the keyword NOT are optional in front of the target.

A relative position consists of the keyword AT or one of the keywords BEFORE or AFTER; a signed number is optional before the latter two keywords.

The target can be a range of characters or a delimited string. A number is optional after the target. A hex range matches any character within the range. The keyword STRING followed by a delimited string matches the string. The keyword ANYOF followed by a delimited string matches any one character in the string. (The keyword is optional before a one character string, because the effect is the same in either case.)

The matching character or string is discarded when records are split at a target. AT is the default qualifier. No parameters means split at blank characters.

Operation: *split* scans the record matching the pattern. When MINIMUM is specified, *split* skips the number of characters specified before it starts looking for the pattern.

Use a number after the pattern to make *split* stop after the pattern has been matched that number of times and write any remaining input data to the output stream; the default is to continue to the end of the record. *split* writes at most $n+1$ records when a number is specified.

A split position is established when the pattern has been matched. With no modifiers, it is before the first character matching the pattern; with the options BEFORE and AFTER, it can

be offset any number of characters to the left or right by coding *snumber*. *n* AFTER a target is equivalent to *m* BEFORE a target, where *m* is $-n-\text{length}(\text{target})$. When a split position is established within or after the record, a record is written with data from the previous split position (initially before the first character in the record) to the newly established split position or the end of the record, whichever occurs first. When splitting AT, the split position is updated with the length of the target after a record is written, so that the target is discarded; thus, records that consist entirely of the target are discarded.

Record Delay: *split* does not delay the last record written for an input record.

Premature Termination: *split* terminates when it discovers that its output stream is not connected.

Converse Operation: *join*.

See Also: *chop*, *deblock*, *fblock*, and *spill*.

Examples: To write each blank-delimited word as a separate record:

```
pipe literal a b c | split | console
▶a
▶b
▶c
▶Ready;
```

The set buffer address order (X'11') marks the beginning of a field in an inbound 3270 data stream from a read modified command. If the 3270 data stream uses twelve-bit addressing, you can split each inbound transmission into individual fields by | split before 11 |. This is too simplistic if the 3270 data stream uses fourteen or sixteen bit addressing; the two-byte buffer address that follows the order code could itself contain X'11', which would trigger a split too early. To be sure:

```
... | split minimum 3 before 11 | ...
```

Notes:

1. *split* copies null input records to the output; it does not generate null records.

```
:
:      Caveat emptor! What this means is that a record that contains only the target string,
:      no matter how many instances, is dropped. The pipeline below causes the variable
:      ROB to be dropped.
```

```
:      rob=' '
:      'PIPE var rob | split | var rob'
```

```
:      One way to retain the variable is to strip before splitting; this creates a null record,
:      which is passed by split.
```

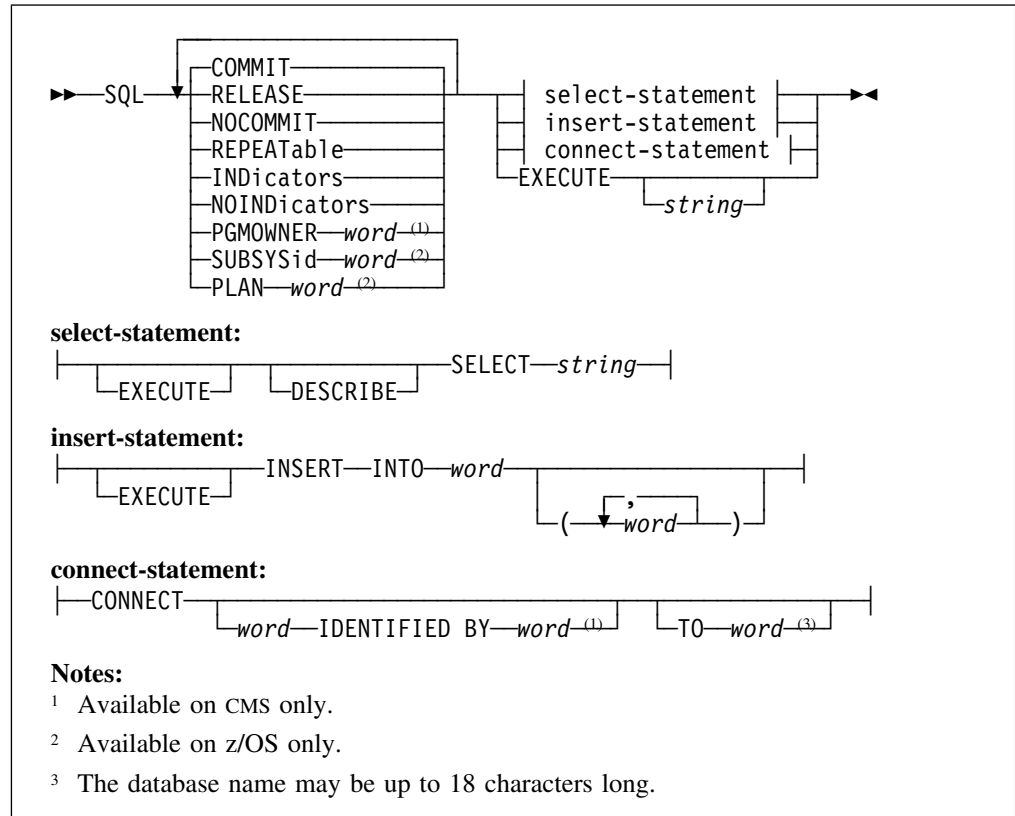
```
:      rob=' '
:      'PIPE var rob | strip | split | var rob'
```

2. Use *deblock* FIXED to split an input record into records of the specified length, where only the last part can be shorter than the record length.
3. The minimum abbreviation of ANYCASE is four characters because ANYOF takes precedence (ANYOF can be abbreviated to three characters).
4. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

sql

sql—Interface to SQL

sql queries DB2 tables, inserts rows into DB2 tables, and issues SQL statements in general.



Type: Device driver.

Syntax Description: 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.

General options:

COMMIT	Commit the unit of work without releasing the connection to the DB2 service machine at completion, or roll back without releasing in the event of an error. This is the default.
RELEASE	Commit work with release at the completion of the stage, or roll back work with release when a negative return code is received from DB2. Use this option when you do not expect to use <i>sql</i> again in the near future; being connected ties up resources in the DB2 virtual machine, but on the other hand there is a certain overhead in reestablishing the connection.
NOCOMMIT	Do not commit the unit of work when processing is complete without errors. Roll back without release in the event of an error. Use this option when processing with multiple cursors or if you wish to issue SQL statements from multiple invocations of <i>sql</i> as a single unit of work. The connection to the DB2 server is retained.

REPEATABLE	Read repeatable isolation option is requested. The default is to use cursor stability. This option is ignored on z/OS; the program isolation is specified when the plan is bound.
INDICATORS	The data streams used by <i>sql</i> SELECT and <i>sql</i> INSERT include indicator halfwords in front of the field data proper. INDICATORS is the default for <i>sql</i> SELECT.
NOINDICATORS	The data streams used by <i>sql</i> SELECT and <i>sql</i> INSERT do not include indicator halfwords in front of the field data proper. For <i>sql</i> 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 field format. NOINDICATORS is the default for <i>sql</i> INSERT.
PGMOWNER	The following word specifies the owner of the access module to use. The default is a configuration variable; see below. The option applies to the particular invocation of <i>sql</i> . (CMS only.)
SUBSYSID	The following word specifies the subsystem identification of the DB2 system to be used. The default is DSN. SSID is a synonym for SUBSYSID. (z/OS only.)
PLAN	The following word specifies the plan to use. The default is FPLSQI; your installation may have specified a different default. (z/OS only.)

Operation: Tables are loaded using *sql* INSERT, queried with *sql* SELECT, and maintained with *sql* EXECUTE.

Select: Perform a query. The argument specifies a complete SELECT statement. One record is written for each row of the result of the query. By default, each column is preceded by a 2-byte indicator word specifying whether the column has a null value or contains data. Use NOINDICATORS to suppress this field in the output record.

In an indicator word, binary zero indicates that the column has a value; a negative indicator word indicates that the column is null. A positive value in the indicator word means that the column is truncated; this should not occur, as 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 */
'pipe sql select * from test where name="Oscar" |...'
```

When *sql* SELECT or *sql* DESCRIBE SELECT is issued with EXECUTE, output from the first query is written to the primary output stream, the result of the second query goes to the secondary output stream, and so on, until there are no further output streams; the result of the remaining queries is written to the highest numbered stream defined. The streams must be connected.

Describe select: The argument is a query to be described. One record is written for each field of the query. Refer to the description of the SQLDA in *DB2 Server for VSE & VM Application Programming*. Each record has five blank-delimited fields of fixed length:

- 3 The decimal number defining the field type.
- 16 The field type decoded, or "Unknown" if the field type is not recognised by *CMS Pipelines*. 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.
- 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.
- 30 The field name. The record is truncated at the end of the name; the name field is from 1 to 30 bytes.

Multiple queries are performed as described with SELECT above.

```
/* Describe the result of a query */
'pipe sql describe select * from test | console'
```

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.

The remainder of this section describes how *sql* processes an insert statement that has no values() clause and no subquery. This is supported only on CMS because DB2 does not provide the underlying interface to insert on a cursor. When no values() clause is specified, *sql* supplies one that references fields in input records. A row is inserted into the table for each input record. You can specify a list of fields to insert in parentheses; this list is used by *sql* to build a data area describing the input record.

When there is no list of columns after the name of the table, the input record contains data for all columns in the table in the order returned by *sql* DESCRIBE SELECT * FROM. When a list of columns is specified, the input record has the columns in the order specified and in the format returned by the describe function. You cannot insert a literal value into a column; use *spec* to put a literal into all input records. The format of an input record is the same as the output record from *sql* SELECT. In particular, a variable length column must be padded to its maximum length.

Use the option INDICATORS when you wish to load null values into selected columns. All columns must have a 2-byte prefix, which is binary zero when the field has a value; it is negative to indicate a null value. The default is not to use indicator words.

Input records are read from the primary input stream when EXECUTE is omitted. When *sql* INSERT statement(s) with no values() clause are issued with the EXECUTE option, the first statement reads records from the secondary input stream, the second statement reads from stream number 2, and so on. It is an error if a stream is not defined.

```
/* Insert data into a table */
'pipe < to insert | sql insert into test' /* CMS ONLY */
```

Connect: Connect to a database. On VM, you can specify the user ID and password under which you wish to communicate with DB2. The first *word* represents the user ID; the second one represents the password. The keyword TO specifies that you wish to connect to a particular database. The database name can be up to 18 characters long.

Execute: Perform SQL statements. A statement after EXECUTE is issued first; the primary input stream is then read and each record is performed. All SQL statements are performed as a single unit of work. Most SQL statements are supported; refer to the description of the PREPARE statement in *DB2 Server for VSE & VM Application Programming*, 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 SQLCODE -515. Processing stops as soon as an error is reported by DB2.

```
/* Drop a program */
'pipe literal drop program pipsqi | sql execute'
```

Using Multiple concurrent sql stages: Up to ten *sql* stages can run concurrently in all active pipelines. It is paramount that the option NOCOMMIT be used. 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 using their cursors. Explicit commit or rollback is done with *sql* COMMIT and *sql* ROLLBACK.

```
/* Merge two tables */
'PIPE (end ?)',
  'sql nocommit select * from table1 | p:... ',
  '?sql nocommit select * from table2 | p:'

If RC/=0
  Then commit='rollback'
  Else commit='commit'

'PIPE sql execute' commit 'work release'
```

Streams Used: SQL statements are read from the primary input stream when EXECUTE is used. On CMS, rows to insert are read from the input streams. The result of queries is written to the output streams.

Record Delay: *sql* produces all output records from a query before it consumes the corresponding input record.

Commit Level: *sql* starts on commit level -4. It connects to the database engine, allocates a cursor, and then commits to level 0.

Premature Termination: *sql* terminates when it discovers that any of its output streams is not connected. It also terminates if a negative return code (indicating an error) is received from DB2; the unit of work is rolled back when a negative return code is received, unless DB2 indicates it has already done so.

See Also: *sqlcodes* and *sqlselect*.

Notes:

1. *spec* is often used to insert indicator words for columns that are always present.
2. On CMS, use SQLINIT to specify the database to access, before using *sql* to access it.
3. The result of a query can be a four byte binary integer; use *spec* to convert it to decimal, if desired.

```
/* Determine query size */
'PIPE',
  ' sql select count(*) from test where name="Oscar" ',
  '| spec 1-* c2d 1',
  '| var rows'
if RC=0 then if rows>0 then call process
```

4. The access module must be generated before you can access DB2 tables with *CMS Pipelines*. DMSPQI ASMSQL is the input to the preparation process. Your database administrator must give connect privileges to the user DMSPQI and specify the password (the examples use “wrench”).

```
grant connect to dmspipe identified by wrench
```

From the user ID used to maintain *CMS Pipelines*, run the REXX program shown here after you have issued SQLINIT to establish connection to DB2. The program generates the access module and grants public use of it.

```
/* Generate access module */
EXEC SQLPREP ASM PP(PREP=DMSPQI',
                    'USER=DMSPIPE/WRENCH',
                    'NOPUNCH BLOCK ISOL(USER))',
                    'IN(DMSPQI)')
if RC/=0 then exit RC
'PIPE',
  ' literal grant run on DMSPQI to public',
  '| literal connect dmspipe identified by wrench',
  '| sql execute'
exit RC
```

Discard the resulting ASSEMBLE file.

Do not put quotes around the user ID when you issue the CONNECT statement through *sql*

5. *sql* supports DB2 on z/OS. Use the option SUBSYSID to specify the DB2 resource you wish to use if different from the default (DSN). *sql* issues a Call Attachment Facility (CAF) OPEN call to connect to DB2. The RELEASE option causes *TSO Pipelines* to issue a CLOSE call when processing is complete. Make sure all requests are issued by the same task. This is best done by issuing subroutine pipelines from a stage that is written in REXX. If, however, several PIPE commands must be issued, it is *de rigueur* to use Address LINK rather than TSO commands to issue pipeline specifications.
6. Because DB2 does not support insert on a cursor, *sql* INSERT must have a values() clause specifying literals on z/OS. Use *spec* to construct an insert statement from data in the record.
7. To access a z/OS database through distributed relational access you must export the plan.

```
SQLINIT DB(VM_DB) PROTOCOL(AUTO)
FILEDEF PLAN DISK PIPE BIND A4
SQLDBSU
UNLOAD PACKAGE(DMSPIPE.DMSPQI) OUTFILE(PLAN);
CONNECT TO MVS_DB;
RELOAD PACKAGE(DMSPIPE.DMSPQI) REPLACE KEEP INFILE(PLAN);
EXIT
```

Note the semicolons after each statement in SQLDBSU. The assumptions are:

- VM_DB is the name of your local DB2 database on the system where you are running *CMS Pipelines*.
- MVS_DB is the name of the DB2 for z/OS database.
- You can already connect from VM to z/OS. (That is, your UCOMDIR NAMES is already set up.)
- You are using SQLDBSU Version 3.1 or later.
- You have bind authority in the destination z/OS database.

Return Codes: Error codes from DB2 are reflected in the return code; such return codes are negative. Positive return codes represent errors detected by *CMS Pipelines*. When DB2 returns a positive number that is not 100 (which means “no more data”), *CMS Pipelines* generates an error message and terminates.

Configuration Variables: On CMS, two configuration variables supply the default program owner and the default program name to be used by *sql*.

SQLPGMOWNER specifies the program owner; the default is 5785RAC in the PIP style; it is DMSPICE in other styles.

SQLPGMNAME specifies the program name; the default is PIPSQI in the PIP style; it is DMSPQI in other styles.

sqlcodes—Write the last 11 SQL Codes Received

sqlcodes writes a 44-byte record with the last 11 SQLCODEs received. It is used by *help* SQLCODE.

▶▶—SQLCODES—◀◀

Type: Arcane device driver.

Placement: *sqlcodes* must be a first stage.

Output Record Format: 44 bytes are written to the primary output stream; these are 11 fullwords with 4 bytes for each nonzero (other than 100) return code expressed as a binary number in two’s complement notation. The return code received most recently is in the last four bytes; the oldest is in the first four bytes. The leftmost slots contain zero when fewer than 11 nonzero return codes have been received from SQL.

See Also: *sql* and *help*.

sqlselect—Query a Database and Format Result

sqlselect issues an SQL query and converts the result to printable format with a header line showing the names of the columns.

▶▶—SQLSELECT—┌(—| SQL Options |—)┐ | Select Operands |▶▶

Type: Device driver.

Placement: *sqlselect* must be a first stage.

Syntax Description: If the first non-blank character is a left parenthesis, the string up to the first right parenthesis is processed as options; refer to the description of the *sql* built-in program. The remainder of the argument is processed as an SQL Select statement.

Operation: *sqlselect* obtains a description of the query from SQL, computes a pipeline that will convert the query result to printable text, writes a heading line showing the names of the columns in the output file, and then performs the query with formatting.

stack

Output Record Format: It issues a subroutine pipeline to describe the query. This will cause a commit to level 0 if the query can be described.

Commit Level: *sqlselect* starts on commit level -4.

Premature Termination: *sqlselect* terminates when it discovers that its output stream is not connected.

See Also: *sql*.

Examples: To query one of the sample databases:

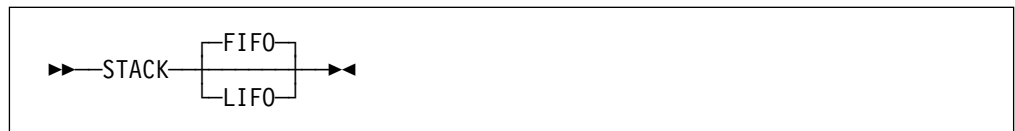
```
pipe sqlselect salary, name from staff where years is null | console
```

Notes:

1. Timestamps cannot be formatted, because their encoding is not published. Such fields will display as apparently random alphanumeric characters.
2. *sqlselect* uses *sql* under the covers; the SQL configuration variables apply to *sqlselect* as well.

stack—Read or Write the Program Stack

When *stack* is first in a pipeline, it reads lines from the console stack into the pipeline. When *stack* is not first in a pipeline, it copies the lines in the pipeline onto the program stack.



Type: Device driver.

Syntax Description: No argument is allowed when *stack* is first in a pipeline. A keyword is optional when *stack* is not a first stage.

Operation: When *stack* is first in a pipeline, it issues the command `SENTRIES` on CMS to obtain the number of lines on the console stack; it then reads as many lines from the stack as indicated by the return code and writes these lines to the output stream. The intent is to be able to drain the stack into the pipeline, including null lines that would make *console* stop. A terminal read may result if another stage (possibly another invocation of *stack*) reads from the stack concurrently with *stack*.

When *stack* is not first in a pipeline, records on the input stream are stacked and then copied to the output stream. By default the lines are queued FIFO in the CMS console stack. Beware of loops if lines are being read by *console* at the beginning of the pipeline; such loops are best prevented with a *buffer* stage.

Streams Used: *stack* passes the input to the output.

Record Delay: *stack* strictly does not delay the record.

Premature Termination: When it is first in a pipeline, *stack* terminates when it discovers that its output stream is not connected. An additional line may or may not have been consumed from the stack when this is discovered.

Examples: The contents of the stack may be saved in REXX variables while running a REXX program and a new stack created at the end of the program (see the following example), but the effect of any MAKEBUF is lost.

```
/* Save the stack */
'PIPE',
  'stack|',
  'stem save_stack.'

/* Process */

/* Restore saved stack */
'PIPE',
  'stem save_stack.|',
  'stack'
```

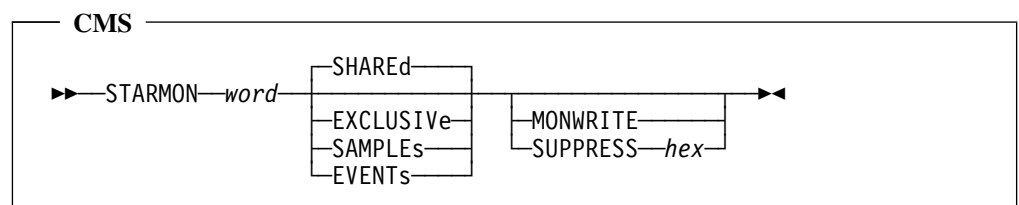
Notes:

1. Due to the limited width of the CMS stack, stacked data are truncated after 255 characters.

starmon—Write Records from the *MONITOR System Service

starmon connects to the *MONITOR system service and writes the data it receives into the pipeline. The records are logical records, beginning with the twenty bytes prefix defined for monitor records. You can elect to write only sample data or event data and you can suppress records from one or more domains.

Before invoking *starmon*, you should attach the monitor segment to the virtual machine using the CMS command “segment load” and also enable the monitor domains you wish to process using the CP command “monitor”.



Type: Arcane host command interface.

Placement: *starmon* must be a first stage.

Syntax Description:

word Specify the name of the monitor shared segment to be used. The segment must have been attached to the virtual machine before *starmon* is invoked.

The first keyword specifies the type of interface used to the system service.

SHARED SHARED is the default. It implies both EVENTS and SAMPLES.

EXCLUSIVE Connect to the monitor service with exclusive use of the monitor segment. *starmon* writes sample and event data as enabled by the MONITOR command.

starmon

SAMPLES	Connect to the monitor service in shared mode. Only sample data are to be retrieved from the monitor segment.
EVENTS	Connect to the monitor service in shared mode. Only event data are to be retrieved from the monitor segment.
MONWRITE	Write the monitor data in the same format as produced by the MONWRITE utility.
SUPPRESS	Specify a bit map of the monitor domains you wish to suppress. The next word is converted from hexadecimal to binary. The sixteen right-most bits are used as a mask. Thus, SUPPRESS 8000 specifies that <i>starmon</i> should not write records from the system domain. Performance will improve if you enable monitor domains selectively using the MONITOR command rather than using the SUPPRESS option to ignore data from enabled domains.

Operation: *starmon* connects to the CP *MONITOR system service using the Inter User Communication Vehicle (IUCV). The message limit is zero, which selects the default for the service requested. The user parameters (IPUSER) are set according to the options specified.

starmon sets up the immediate command HMONITOR. Issue this command to halt the *starmon* stage.

starmon does not complete normally.

Output Record Format: Monitor records as defined in the MONITOR LIST1403 sample file. In contrast to the MONWRITE command, *starmon* writes each monitor record as a separate logical record. The logical record begins with the MRHDR structure.

Commit Level: *starmon* starts on commit level -2000000000. It verifies that no other stage has requested a connection to the *MONITOR service, sets up an immediate command (HMONITOR), connects to the system service, and then commits to level 0.

Premature Termination: *starmon* terminates when it discovers that its output stream is not connected. *starmon* terminates when CP signals that it has not processed the data in time. This is accompanied by error messages indicating a nonzero IPAUDIT field. This indicates that CP has changed the monitor data under *starmon*; the integrity of the output from the pipeline is questionable.

starmon terminates when the immediate command HMONITOR is issued while it is waiting for CP to provide a new batch of monitor records. *starmon* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *starmsg* and *starsys*.

Examples:

The following REXX program to demonstrate extracting information from the monitor data. The MONRED REXX is not provided with *CMS Pipelines*.

Syntax Description: The arguments are optional. A word beginning with an asterisk may be followed by a command string.

To reach a service other than *MSG, specify as the first operand the name of the CP service required, beginning with an asterisk (for example, *MSGALL). You can connect to any system service that sends messages and does not require a reply. *MSG is the default.

Operation: *starmsg* connects to a CP system service using the Inter User Communication Vehicle (IUCV). The message limit is zero, which selects the default for the service requested. The user parameters (IPUSER) are set to binary zeros. Message data in the parameter list are not supported.

starmsg sets up an immediate command that may be used to halt the *starmsg* stage. The name of the immediate command is the name of the service prefixed by an 'H' (for instance, HMSG).

If it is present, the command string is sent to the CMS subcommand environment after *starmsg* is connected to the system service.

When *starmsg* is not first in a pipeline, it issues each command to CMS. When the command is complete, *starmsg* loops writing any responses trapped to the output and suspending itself to let these responses be processed. When no more responses arrive, *starmsg* tries to read the next input record. It disconnects from the system service and terminates normally when it receives end-of-file on the input. When *starmsg* is first in a pipeline, it does not terminate normally.

Output Record Format: Columns 1-8 contain the message class (IPTRGCLS) converted to hexadecimal (eight bytes). The message class is the only field from the interrupt parameters that is present in the output record. (Refer to *z/VM CP Programming Services*, SC24-6272, for the authoritative meaning of the message class; some common message classes are shown below.) The message follows (as received with IUCV RECEIVE). For message classes 1, 2, 4, and 8, the first eight bytes of the message (columns 9-16 of the output record) contain the user ID of the sending virtual machine, padded with blanks.

<i>Figure 400 (Page 1 of 2). Often Used Message Classes</i>		
Class	Enable	Message Source
1	MSG	Messages sent by the CP command “message” or the CP command “msgnoh”.
2	WNG	Warnings sent by the CP command “warning” command.
3	CPCONIO	Output from CP commands that are issued by the virtual machine (unless the command is issued by diagnose 8 with a response buffer—see <i>cp</i>). Other CP output that is not covered (or enabled) by the other message classes.
4	SMSG	Special messages sent by the CP command “smsg”.
5	VMCONIO	Virtual machine output to the console. For example, data displayed by the REXX Say instruction.
6	EMSG	CP error messages.
7	IMSG	CP informational messages.

Figure 400 (Page 2 of 2). Often Used Message Classes

Class	Enable	Message Source
8		Single console image facility output from a virtual machine that has identified the machine running <i>starmsg</i> as its secondary user. These messages cannot be redirected to the console once <i>starmsg</i> *MSG has connected to the message system service.

Streams Used: Records are read from the primary input stream and written to the primary output stream.

Record Delay: The delay is unspecified when *starmsg* is not a first stage.

Commit Level: *starmsg* starts on commit level -2000000000. It verifies that no other stage has requested a connection to the CP service, sets up an immediate command environment, establishes the connection to the CP service, and then commits to level 0.

Premature Termination: *starmsg* terminates when it discovers that its output stream is not connected. A particular invocation of *starmsg* is terminated when the immediate command exit is driven that has the name of the service with an 'H' substituted for the leading asterisk; for instance, HMSGALL. *starmsg* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *starmon* and *starsys*.

Examples: *starmsg* may be used as an alternative to the programmed operator. With such an application, you may wish to stop the application by sending it a particular message:

```
/* Listen for messages */
'PIPE starmsg | tolabel 00000001'left(userid(),8)'STOP| ...
```

Then, when you issue the CP command “message * stop”, *starmsg* writes an output line that causes the *tolabel* stage to terminate. This severs *starmsg*'s output stream which in turn causes *starmsg* to terminate. You must have issued the CP command “set msg iucv” for the message to be trapped by *starmsg*; the *tolabel* stage is all in vain if the message does not get trapped.

To issue a single command and then terminate:

```
cp set cpconio iucv
pipe literal RELEASE C ( DET | starmsg | hole

cp set msg iucv
pipe literal CP SMSG RSCS Q SYS A | starmsg | ...
pipe hole | starmsg CP SMSG RSCS Q SYS A | ...
```

Still, the probability is low that RSCS will respond before *starmsg* discovers that it has no more commands to issue. To terminate *starmsg* after five seconds:

```

/* Wait for a short while */
'PIPE (end ? name STARMSG)',
  '|literal +5',           /* Five seconds          */
  '|delay',               /* Wait a bit           */
  '|g: gate',            /* Shut the gate        */
  '?starmsg CP MSG RSCS Q SYS A', /* Trap responses      */
  '|g:',                 /* Until the interval expires */
  '|...'

```

When the record is written by *delay* five seconds after it reads it, the output stream from *starmsg* is severed. This will cause *starmsg* to terminate.

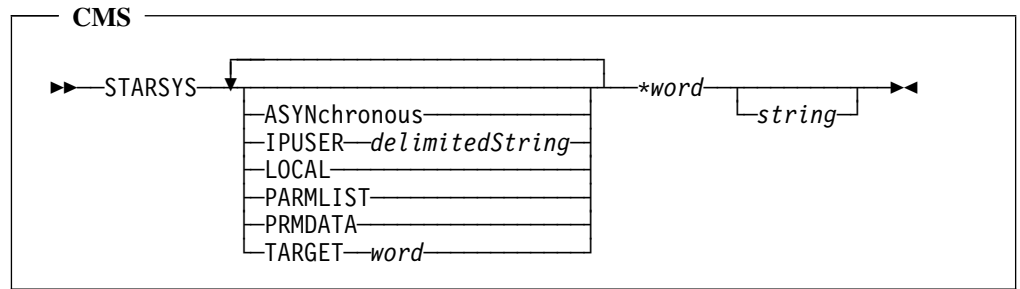
Notes:

1. The CMSIUCV macro is used to connect to the message service.
2. Use CP SET to select which responses you wish to process; for instance, “cp set cpconio iucv”.
3. When CP CONIO is set to IUCV, all CP console output is presented through the *MSG interface. Enable IUCV for other settings to make it easier to distinguish different forms of CP console output. For instance, messages are presented as CP console output (message class 3) when the message setting is ON, but as messages (message class 1) when the MSG setting is set to IUCV.
4. Any CP system service can be selected; results are unpredictable when the service is not a message service or a similar one-way service.
5. You cannot connect to *MSGALL when CMS FULLSCREEN is ON because CMS is already connected to the service; CP rejects further attempts to connect.
6. A virtual machine can have only one *starmsg* or *starsys* stage at a time for a particular system service.
7. Though it is possible to use *starmsg* to connect to the *ACCOUNT, *LOGREC, and *SYMPTOM system services, the recommended device driver is *starsys*. *starmsg* may require large amounts of buffer space to hold all pending messages, and once a message is read by *starmsg*, it is purged by CP. *starsys*, on the other hand, accepts only one message from CP at a time and does not signal to CP that the message has been received until after the corresponding output record has been consumed.
8. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

starsys—Write Lines from a Two-way CP System Service

starsys connects via the Inter User Communication Vehicle (IUCV) to a two-way system service (*ACCOUNT, *LOGREC, or *SYMPTOM) to retrieve system information and write it into the pipeline.

When *starsys* is not first in a pipeline, it reads replies to CP from its input.



Type: Arcane host command interface.

Syntax Description: Options may be specified when *starsys* is not first in a pipeline. After these, a word beginning with an asterisk may be followed by a command string.

ASYNCHRONOUS The writing of messages is independent of the reading replies. This allows for multiple requests being processed simultaneously. Whether CP will allow multiple concurrent messages is another matter. ASYNCHRONOUS implies PARMLIST, since the interrupt parameter identifies the message to reply to. When ASYNCHRONOUS is omitted, there can be only one message in the pipeline at a time; *starsys* knows the message identifier to use in the reply.

IPUSER Specify a delimited string. The string is truncated after sixteen bytes. The delimited string specifies the value to pass in the IPUSER field of the IUCV connection request. The field is initialised to binary zeros, except that the two-way bit is set in offset 8. Only as many bytes as specified in the delimited string are copied to the IPUSER field. Thus the two-way flag is cleared only when more than eight bytes are specified.

LOCAL Set the ILOCAL bit to limit the search for system services to the current system.

PARMLIST Prefix the 28-byte interrupt parameter list to the message received. The first 28 bytes of the input record are taken to be the reply parameter list. When the PRMDATA flag is off, the remainder of the input record (if any) is sent as the reply.

PRMDATA Set the IPRMDATA flag in the connect parameter list. Some services require this flag to be specified; others require it to be omitted.

TARGET Specify a word. The word is translated to upper case and inserted in the IPTARGET parameter on connect.

***word** Specify the name of the CP service required, beginning with an asterisk (for example, *LOGREC). When *starsys* is first in the pipeline, you can connect to any CP service that sends messages and expects a reply: *ACCOUNT, *LOGREC, *SYMPTOM.

Operation: *starsys* connects to a CP system service using the Inter User Communication Vehicle (IUCV). The message limit is zero, which selects the default for the service requested. The user parameters (IPUSER) are set to binary zeros except the byte at offset 8 which is set to X'02', indicating that the two-way protocol is desired. Message data in the parameter list are not supported unless *starsys* is not a first stage.

starsys sets up an immediate command that may be used to halt the *starsys* stage. The name of the immediate command is the name of the service prefixed by an 'H' (for instance, HACCOUNT).

If it is present, the command string is sent to the CMS subcommand environment after *starsys* is connected to the system service.

When *starsys* is first in the pipeline, it sends a reply to the message as soon as the output record is consumed.

When *starsys* is not first in a pipeline, it sends a reply when a record is available on the input. The record is discarded.

starsys does not complete normally.

Commit Level: *starsys* starts on commit level -2000000000. It verifies that no other stage has requested a connection to the CP service, sets up an immediate command environment, establishes the connection to the CP service, and then commits to level 0.

Premature Termination: *starsys* terminates when it discovers that its output stream is not connected. A particular invocation of *starsys* is terminated when the immediate command exit is driven that has the name of the service with an 'H' substituted for the leading asterisk, for instance HACCOUNT. *starsys* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *starmon* and *starmsg*.

Examples:

To process accounting data on z/VM to extract type 4 accounting records (potential hacker activity):

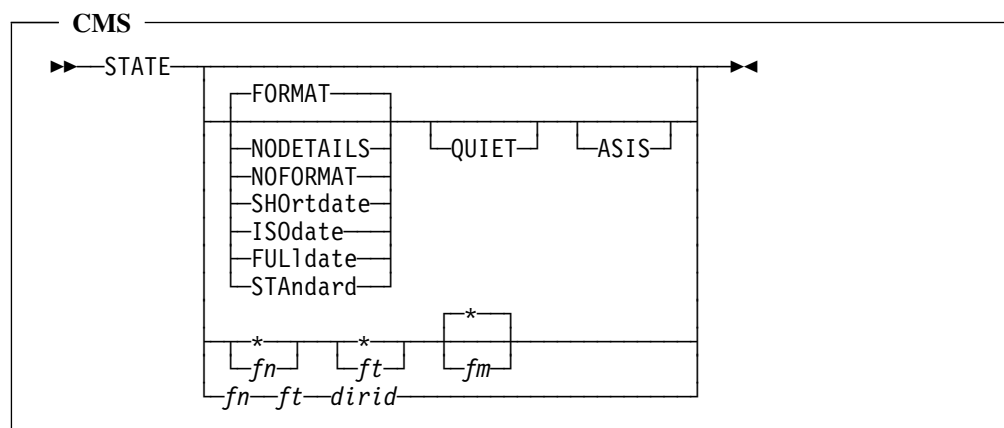
```
/* Account to a file */
'CP RECORDING ACCOUNT ON LIMIT 255'
'PIPE (name STARSYS)',
  '|starsys *account',
  '|locate 80 /4/',
  '|spec /Hacker afoot? / 1 1.8 next 29-32 nextword 71-78 nextword',
  '|console'
If Userid() <> 'OPERACCT'
  Then 'CP RECORDING ACCOUNT OFF PURGE QID' Userid()
```

Notes:

1. The CMSIUCV macro is used to connect to the system service.
2. Any CP system service can be selected; results are unpredictable when the service is not a two-way service.
3. A virtual machine can have only one *starmsg* or *starsys* stage at a time for a particular system service.
4. *starsys* is still unsuitable for services to which the program must send, such as *SPL.
5. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

state—Provide Information about CMS Files

state writes information about CMS files on a minidisk or in a Shared File System (SFS) directory.



Type: Device driver.

Syntax Description: Arguments are optional. The argument string can consist of keywords or the name of a file.

FORMAT	Information about files that are found is written in a printable format using the short date format.
NOFORMAT	The raw control block describing a file is written.
NODETAILS	The file name as specified is written.
QUIET	Set return code 0 even when one or more files are not found; the default is to set return code 28 or 36 when files are not found.
FULLDATE	The file's timestamp is formatted in the American format, with the century: 3/09/1946 23:59:59.
ISODATE	The file's timestamp is formatted with the century in one of the formats approved by the International Standardisation Organisation: 1946-03-09 23:59:59.
SHORTDATE	The file's timestamp is formatted in the American format, without the century: 3/09/46 23:59:59.
STANDARD	The file's timestamp is formatted as a single word in a form that can be used for comparisons: 19460309235959.
ASIS	Do not look for files with the name and type in upper case after it is determined that the file does not exist with the name and type as written.

Alternatively, you can specify a file in the same format as an input record.

Operation: When a file argument is specified, the first output record contains information about the specified file. Then a line is generated for the file specified on each input line read.

Each file is processed as follows: The third word (file mode, name definition, or directory) is translated to upper case. All accessed minidisks and directories are searched if the third word is omitted or is an asterisk. *state* first looks for a file that has the file name and file

state

type as written. If the file does not exist with a file name and a file type as entered and ASIS is omitted, the file name and the file type are translated to upper case and the search is retried. If the file is still not found, the file name, as written originally, is written to the secondary output stream (if it is connected).

Input Record Format: Two or three words, specifying the file name, file type, and optionally the file mode, name definition, or directory. When a mode is specified, the file name or the file type, or both, can be specified as a single asterisk, which means that it matches any file; other forms of “wildcards” are not supported by the underlying CMS interface. The underlying interface to look in a directory does not support asterisks.

Output Record Format: The primary output stream: When NOFORMAT is specified, the output record contains 64 bytes in the format defined by the FSTD data area. When NODETAILS is specified, the output record contains the input record (if the file exists).

Otherwise, selected fields of the file status are formatted and written as a record: the file name, type, and mode; the record format and logical record length; the number of records and the number of disk blocks in the file; the date and time of last change to the file.

When the file is in an SFS directory that is not accessed, the file mode is shown as a hyphen (-). When the file is on an accessed mode, the real file mode is shown. Thus, the mode shown may not be the mode specified. When a name definition or a directory is specified and the file resides in SFS, the fully qualified path to the directory that contains the file is appended after the timestamp. (The file can reside on a minidisk that is accessed as an extension to a mode on which the directory is accessed.)

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded. When a file is found, information about it is written to the primary output stream (if it is connected). When a file is not found, the input record (or the argument string) is passed to the secondary output stream (if it is connected).

Record Delay: *state* does not delay the record.

Commit Level: *state* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *state* terminates when it discovers that no output stream is connected. It also stops if it receives a return code from CMS that is neither 0, 28, nor 36.

See Also: *afjfst*, *fjmfst*, and *stew*.

Examples: To show which files in a list do not exist:

```
pipe (end ?) < file list | s:state ? s: | console
```

The primary output stream is not connected; the secondary output stream is connected to the *console* stage.

To test whether a file exists without a CMS error message on failure, use:

```
/* Look for the file */  
'PIPE state' file  
return RC
```

Notes:

1. When looking for a file on a mode, *state* exposes the way the CMS command STATE works. Though it is not specified, the CMS command searches the active file table before it looks for files on the file modes.
2. When testing whether several files exist, and you are interested only in the return code, be sure to specify *hole* to avoid premature termination:

```
'PIPE ... | state | hole'
```

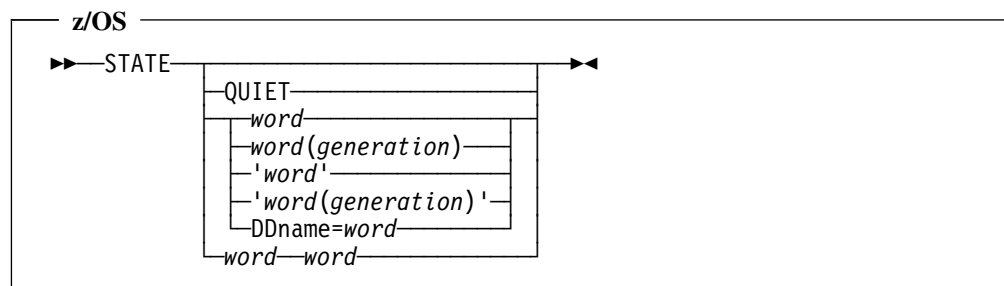
```
if RC=0 then return /* All fine */
```
3. When looking for a file in a directory, *state* exposes the callable services DMSVALDT and DMSEXIST. These interfaces do not distinguish between a file not being present in a directory and a missing directory in the path. Thus, return code 36 is not set for such a file. Refer to *z/VM: CMS Callable Services Reference*, SC24-6259 for information about these interfaces.
4. Be sure to set numeric digits 14 when performing comparisons on STANDARD timestamps; REXX will by default use just nine digits precision. This means that the first digit of the hour will be the least significant one and the remainder of the precision will be lost.
5. SORTED is a synonym for STANDARD.
6. It may be easier to use the CMS STATE command directly if the file is on an accessed mode.

Return Codes:

0	All input lines have been processed. All files exist, the keyword QUIET was specified, or <i>state</i> terminated prematurely.
20	Invalid character in the file name or file type. Processing stops as soon as CMS sets this return code.
24	Invalid file mode. Processing stops as soon as CMS sets this return code.
28	The keyword QUIET was omitted. One or more files were not found; all input lines have been processed.
36	The keyword QUIET was omitted. One or more files referred to a mode that is not accessed; all input lines have been processed.
other	A return code other than 0, 28, or 36 is received from CMS. Processing is terminated with this return code.

state—Verify that Data Set Exists

state reports the data set name (DSNAME) for a file. The file can be specified by DDNAME or data set name.



Type: Device driver.

statew

Syntax Description: Arguments are optional. The keyword QUIET specifies that return code 0 is set also when one or more files are not found; the default is to set return code 28 when files are not found.

Alternatively you can specify the first (or only) file to search for.

Operation: When a DDNAME is specified, SVC 99 is issued to query the allocation. When a DSNNAME is specified, the master catalog is searched for the data set.

Input Record Format: There can be one file name per input record. The name is either a DDNAME (prefixed by the keyword DDNAME=) or a data set name (DSNAME). The current prefix (if any) is prefixed to the DSNNAME unless it is enclosed in single quotes.

Output Record Format: The fully qualified DSNNAME is written to the primary output stream when the file is found. The input record is copied to the secondary output stream when the file is not found.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded. When the file is found, information about it is written to the primary output stream (if it is connected). When the file is not found, the input record (or the argument string) is copied to the secondary output stream (if it is connected).

Record Delay: *state* strictly does not delay the record.

Premature Termination: *state* terminates when it discovers that no output stream is connected.

Examples: To determine whether the DDNAME OSCAR is allocated:

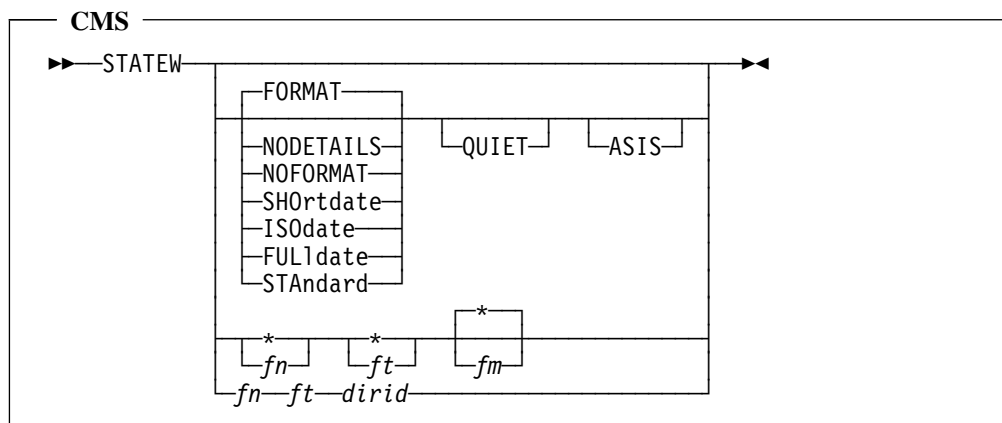
```
pipe state dd=oscar | console
```

Notes:

1. The fact that a data set exists does not imply that it is readable. You may not have RACF authority; the data set could have been migrated.
2. *state* also supports a member name and a DDNAME for which *TSO Pipelines* maintains an open DCB. It is unspecified which DCBs *TSO Pipelines* uses.

statew—Provide Information about Writable CMS Files

statew writes information about writable CMS files on a minidisk or in a Shared File System (SFS) directory.



Type: Device driver.

Syntax Description: Arguments are optional. The argument string can consist of keywords or the name of a file.

FORMAT	Information about files that are found is written in a printable format using the short date format.
NOFORMAT	The raw control block describing a file is written.
NODETAILS	The file name as specified is written.
QUIET	Set return code 0 even when one or more files are not found; the default is to set return code 28 or 36 when files are not found.
FULLDATE	The file's timestamp is formatted in the American format, with the century: 3/09/1946 23:59:59.
ISODATE	The file's timestamp is formatted with the century in one of the formats approved by the International Standardisation Organisation: 1946-03-09 23:59:59.
SHORTDATE	The file's timestamp is formatted in the American format, without the century: 3/09/46 23:59:59.
STANDARD	The file's timestamp is formatted as a single word in a form that can be used for comparisons: 19460309235959.
ASIS	Do not look for files with the name and type in upper case after it is determined that the file does not exist with the name and type as written.

Alternatively, you can specify a file in the same format as an input record.

Operation: When a file argument is specified, the first output record contains information about the specified file. Then a line is generated for the file specified on each input line read.

Each file is processed as follows: The third word (file mode, name definition, or directory) is translated to upper case. All accessed minidisks and directories are searched if the third word is omitted or is an asterisk. *statew* first looks for a writable file that has the file name and file type as written. If the file does not exist with a file name and a file type as entered and ASIS is omitted, the file name and the file type are translated to upper case and the search is retried. If the file is still not found, the file name, as written originally, is written to the secondary output stream (if it is connected).

Input Record Format: Two or three words, specifying the file name, file type, and optionally the file mode, name definition, or directory. When a mode is specified, the file name or the file type, or both, can be specified as a single asterisk, which means that it matches any file; other forms of “wildcards” are not supported by the underlying CMS interface. The underlying interface to look in a directory does not support asterisks.

Output Record Format: The primary output stream: When NOFORMAT is specified, the output record contains 64 bytes in the format defined by the FSTD data area. When NODETAILS is specified, the output record contains the input record (if the file exists).

Otherwise, selected fields of the file status are formatted and written as a record: the file name, type, and mode; the record format and logical record length; the number of records and the number of disk blocks in the file; the date and time of last change to the file.

When the file is in an SFS directory that is not accessed, the file mode is shown as a hyphen (-). When the file is on an accessed mode, the real file mode is shown. Thus, the mode shown may not be the mode specified. When a name definition or a directory is specified and the file resides in SFS, the fully qualified path to the directory that contains the file is appended after the timestamp.

Streams Used: Secondary streams may be defined. Records are read from the primary input stream; no other input stream may be connected. Null and blank input records are discarded. When a file is found, information about it is written to the primary output stream (if it is connected). When a file is not found, the input record (or the argument string) is passed to the secondary output stream (if it is connected).

Record Delay: *statew* does not delay the record.

Commit Level: *statew* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *statew* terminates when it discovers that no output stream is connected. It also stops if it receives a return code from CMS that is neither 0, 28, nor 36.

See Also: *aftfst*, *fmtfst*, and *state*.

Examples: To show which files in a list do not exist:

```
pipe (end ?) < file list | s:statew ? s: | console
```

The primary output stream is not connected; the secondary output stream is connected to the *console* stage.

To test whether a file exists and can be written without a CMS error message on failure, use:

```
/* Look for the file */  
'PIPE statew' file  
return RC
```

Notes:

1. When looking for a file on a mode, *statew* exposes the way the CMS command STATEW works. Though it is not specified, the CMS command searches the active file table before it looks for files on the file modes.

2. When testing whether several files exist, and you are interested only in the return code, be sure to specify *hole* to avoid premature termination:


```
'PIPE ... | statew | hole'
if RC=0 then return /* All fine */
```
3. When looking for a file in a directory, *statew* exposes the callable services DMSVALDT and DMSEXIST. These interfaces do not distinguish between a file not being present in a directory and a missing directory in the path. Thus, return code 36 is not set for such a file. Refer to *z/VM: CMS Callable Services Reference, SC24-6259* for information about these interfaces.
4. Be sure to set numeric digits 14 when performing comparisons on STANDARD timestamps; REXX will by default use just nine digits precision. This means that the first digit of the hour will be the least significant one and the remainder of the precision will be lost.
5. SORTED is a synonym for STANDARD.
6. It may be easier to use the CMS STATEW command directly if the file is on an accessed mode.

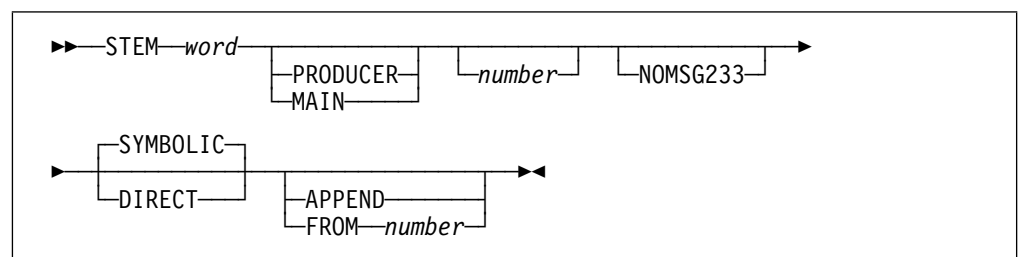
Return Codes:

0	All input lines have been processed. All files exist, the keyword QUIET was specified, or <i>statew</i> terminated prematurely.
20	Invalid character in the file name or file type. Processing stops as soon as CMS sets this return code.
24	Invalid file mode. Processing stops as soon as CMS sets this return code.
28	The keyword QUIET was omitted. One or more files were not found; all input lines have been processed.
36	The keyword QUIET was omitted. One or more files referred to a mode that is not accessed; all input lines have been processed.
other	A return code other than 0, 28, or 36 is received from CMS. Processing is terminated with this return code.

stem—Retrieve or Set Variables in a REXX or CLIST Variable Pool

stem connects a stemmed array of variables to the pipeline. When *stem* is first in the pipeline, the contents of the array are written to the pipeline; an array is built when *stem* is not first in a pipeline.

A stemmed array consists of variables that have names ending in an integer that is zero or positive (the index). The variable that has index 0 contains the count of “data” variables, which are numbered from 1 onward.



Type: Device driver.

Warning: *stem* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *stem* is unintentionally run other than as a first stage. To use *stem* to read data into the pipeline at a position that is not a first stage, specify *stem* as the argument of an *append* or *preface* control. For example, `|append stem ...|` appends the data produced by *stem* to the data on the primary input stream.

Syntax Description: A word is required to specify the stem to fetch or store. It is possible to access a REXX variable pool other than the current one.

The keyword `PRODUCER` may be used when the pipeline specification is issued with `CALLPIPE`. It specifies that the variable pool to be accessed is the one for the stage that produces the input to the stage that issues the subroutine pipeline that contains *stem*, rather than the current stage. (This is a somewhat esoteric option.) To ensure that the variable pool persists as long as this invocation of *stem*, the stage that is connected to the currently selected input stream must be blocked in an `OUTPUT` pipeline command while the subroutine pipeline is running.

The keyword `MAIN` specifies that the REXX variable pool to be accessed is the one in effect at the time the pipeline set was created (either by the `PIPE` command or by the *runpipe* stage). `MAIN` is implied for pipelines that are issued with `ADDDPIPE`.

A number that is zero or positive is optional. It specifies the number of REXX variable pools to go back. That is, *stem* can operate on variables in the program that issued the pipeline specification to invoke *stem* or in one of its ancestors. (When the number is prefixed by either `PRODUCER` or `MAIN`, the variable pool to be accessed is the producer's or the main one, or one of their ancestors.) On CMS, if the number is larger than the number of REXX environments created on the call path from the `PIPE` command, *stem* continues on the `SUBCOM` chain starting with the environment active when `PIPE` was issued.

Specify the option `NOMSG233` to suppress message 233 when the REXX environment does not exist. Either way, *stem* terminates with return code 233 on commit level -1 when the environment does not exist.

The keyword `SYMBOLIC` specifies that REXX should treat the variable names generated as it would a variable that is written in a program. `DIRECT` specifies that REXX should use the variable name exactly as written.

The keyword `APPEND` is optional when *stem* is not a first stage. The keyword `FROM` followed by a number is optional.

When *stem* is first in a pipeline or the `APPEND` keyword is specified, the variable `<stem>0` is read from the variable pool; it must be an integer that is zero or positive.

Operation: When *stem* is first in a pipeline the value of the variable `<stem>0` specifies the number of the last record to write to the pipeline; unless `FROM` is specified to set the starting index number, the first output record contains the value of `<stem>1`, the second record contains the value of `<stem>2`, and so on to the number specified. No record is written if `<stem>0` is zero or less than the value specified after `FROM`.

When *stem* is not first in a pipeline and the keywords `APPEND` and `FROM` are omitted, variables `<stem>1`, `<stem>2`, and so on, are set to the contents of each successive input record. Records are copied to the primary output stream (if it is connected) after the variable is set. When `APPEND` is specified, writing starts with `<stem>n` where `n` is one more than the value returned for `<stem>0`. The index of the last variable set is stored in the

variable <stem>0 at end-of-file. When there are no input records, <stem>0 is left unchanged if APPEND is specified; <stem>0 is set to zero if APPEND is not specified.

Record Delay: *stem* strictly does not delay the record.

Commit Level: *stem* starts on commit level -1. It verifies that the REXX environment exists (if it did not do so while processing its parameters) and then commits to level 0.

See Also: *var* and *varload*.

Examples: To read a file and process it in the reverse order:

```
/* Reverse TYPE */
'PIPE <' arg(1) '|' stem x.'
Do i=x.0 to 1 by -1
  Say x.i
End
```

To transfer an array from the caller:

```
/* Obtain parameters from caller */
address command,
  'PIPE stem parms. 1 | stem parms.'
```

The inverse pipeline can be used to transfer the contents of the array back to the caller:

```
/* Return parameters to caller */
address command,
  'PIPE stem parms. | stem parms. 1'
```

Notes:

1. The APPEND keyword is not the same as the *append* built-in.
2. When a pipeline is issued as a TSO command, IKJCT441 is called to access the variable pool. When the command is issued with Address Link or Address Attach, *stem* accesses the REXX environment from where the command is issued.
3. *CMS Pipelines* maintains a reference to the current variable environment for each stage. Initially this is the environment in effect for the PIPE command with which the original pipeline was started.

When a REXX program is invoked (as a stage or with the REXX pipeline command), its environment becomes the current one, with a pointer to the previous one.

When a pipeline specification is issued with the *runpipe* built-in program or the CALLPIPE pipeline command, the current environment is the one in effect for the stage issuing *runpipe* or CALLPIPE; it is known to persist while the subroutine pipeline runs. On the other hand, when a pipeline specification is issued with the ADDPIPE pipeline command, the stage that issues ADDPIPE runs in parallel with the added pipeline specification; it can terminate at any time (indeed, even before the new pipeline specification starts running). Therefore, for ADDPIPE, the current environment is set to the one for the last *runpipe* or the one at initial entry on the PIPE command. Thus, the MAIN option has effect only for pipeline specifications that are issued by the CALLPIPE pipeline command.

4. Unless DIRECT is specified, *stem* uses the symbolic interface to access REXX variables. This means that you should write the variable name the same way you would write it in an assignment statement. Consider this program fragment:

stfle

```
/* Process an array */  
x='fred'  
'PIPE literal a | stem z.x.'
```

The variable Z.fred.1 is set to 'a '. On the other hand, this would set the variable Z.x.1:

```
/* Process directly */  
'PIPE literal a | stem Z.x. direct'
```

Note that the stem must be in upper case when DIRECT is used.

5. An unset variable (that is, a variable that has been dropped or has never been assigned a value) is treated differently by the three variable repositories: REXX returns the name of the variable in upper case; EXEC2 and CLIST return the null string.

6. It is unspecified how many variables *stem* obtains at a time from the variable pool. Applications that update a stemmed array to add items to it should buffer the file before it is written back to the array:

```
'pipe stem x. | dup | buffer | stem x.'
```

Without the buffering, variable x.2 could be created (containing a copy of the contents of variable x.1) by the second *stem* stage before the first stage has read it.

Applications should not rely on this behaviour of *stem*.

7. For REXX stems, it is normal to specify a period as the last character of the stem (the first word of the argument string). To allow access to EXEC2 variable pools, *stem* does not append a period to the word specified. This means that you can use *stem var* to set simple variables, such as VAR1, VAR2, and so on. VAR0 will be set to the count of variables set.

! *stfle*—Store Facilities List

! *stfle* writes a single record in which the individual bits indicate the facilities installed in the
! configuration.

! 

! **Type:** Arcane device driver.

! **Placement:** *stfle* must be a first stage.

! **Output Record Format:** The output consists of a single record of one or more binary
! double words produced by the STFLE instruction. Each bit in the output record corresponds
! to a facility assigned to that bit position. Refer to *z/Architecture Principles of Operation*
! for the list of assigned facility bits.

! **Premature Termination:** *stfle* terminates when it discovers that its output stream is not
! connected.

! **Examples:** A list of installed facilities can be produced by numbering the bits in the *stfle*
! output record:

```

! pipe stfle | spec 1-* c2b | fblock 1 | ...
! ... spec 1 1 number from 0 n.4 r | find 1 | substr 2.4 |
! ... join 15 | console
! ▶ 0 7 8 9 10 12 14 15 16 17 18 19 20 21 22 23
! ▶ 24 25 26 27 28 30 31 32 33 34 35 37 38 40 41 42
! ▶ 43 44 45 47 48 49 50 51 52 53 54 55 57 58 59 61
! ▶ 73 74 75 76 77 80 81 82 128 129 131 133 134 135 146 147
! ▶ 156 168
! ▶Ready;

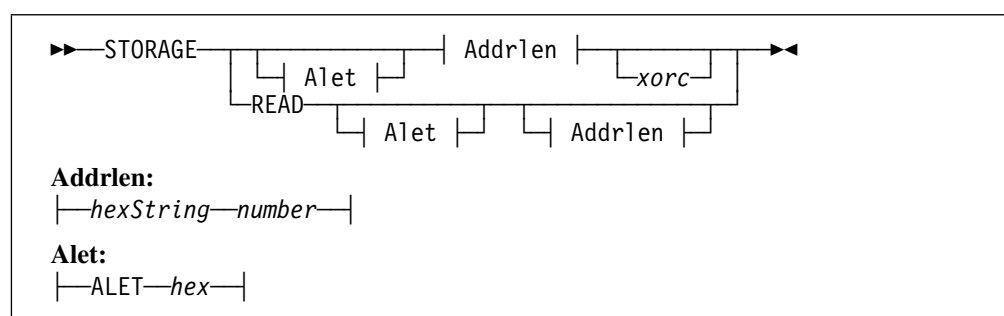
```

Notes:

1. The bitmap observed in a virtual machine may be different from the bitmap presented to CP by the real hardware. For instance, with z/VM running on z/Architecture hardware, a virtual machine running CMS will find bit 2 reset to indicate ESA/390 mode; under z/CMS the bit is set to indicate z/Architecture mode.

storage—Read or Write Virtual Machine Storage

storage connects virtual storage to the pipeline. It copies the contents of virtual storage into the pipeline when it is first in a pipeline or when READ is specified; otherwise it copies records from the pipeline into storage. The storage area may be in your own virtual machine's primary space, a data space you created, or in someone else's shared address space.



Type: Arcane device driver.

Warning: *storage* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *storage* is unintentionally run other than as a first stage. To use *storage* to read data into the pipeline at a position that is not a first stage, specify *storage* as the argument of an *append* or *preface* control. For example, |append storage ...| appends the data produced by *storage* to the data on the primary input stream.

Syntax Description: Two arguments are required, a hexadecimal string and a decimal number. When *storage* is not first in a pipeline and READ is omitted, a third word is required to specify the protect key of the storage area. The key is in the leftmost four bits of a character; the rightmost four bits must be zero. Key zero is rejected. The third operand has no effect on z/OS; specify 80 to be consistent with the CMS implementation. When *storage* is first in a pipeline or READ is specified with an address and length, it ensures that the first and last byte of the storage area are addressable. (On CMS, it only performs this check if the storage area ends beyond the size of the virtual machine unless ALET is specified.) When it is not first in a pipeline and READ is omitted, *storage* verifies that it can modify the first and last byte of the storage area. Areas outside the virtual machine can be specified, but a subsequent stage referencing the contents of the record

storage

sent in the pipeline fails with an addressing exception if part of the storage area is not attached to the virtual machine.

Operation: The arguments are converted to binary and used as the address and length of an area of virtual storage.

When *storage* is first in the pipeline or READ is specified, the address and length are used in an output call, in effect writing virtual machine storage into the pipeline. The area is first copied into your primary space when ALET is specified for a read request.

When READ is specified, *storage* first writes that storage area specified by the address and length, if any, and then a record for each non-blank input record. The record specifies the address (hexadecimal) and length (decimal); the addressability of this area is not verified by *storage* (but it will be by whatever processes the record).

Otherwise, when *storage* is not first in the pipeline, input records are copied into the area in storage. The last part of the record is not copied if the input record is longer than the length of the storage area. The input record is copied to the output, if it is connected.

Streams Used: When *storage* is first in a pipeline and READ is omitted, it writes a record to the primary output stream. When it is not first in the pipeline and READ is omitted, it copies the input record to the output after its contents have been copied into storage.

Record Delay: *storage* strictly does not delay the record.

See Also: *adrspc* and *alserv*.

Examples: To display some possibly not randomly chosen bytes from storage:

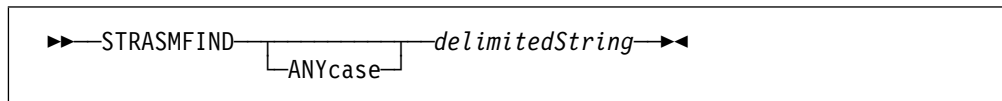
```
pipe storage 200 32 | console
▶z/VM V6.4.0    2019-07-24 16:40
▶Ready;
```

Notes:

1. *storage* can cause message 530 to be issued (destructive overlap) if the storage area overlaps a buffer used by a filter later in the pipeline.
2. Writing to storage has deliberately been made different from reading from storage; this insures against accidental misplacement of a *storage* stage in a pipeline.
3. On CMS, the virtual machine must be in XC mode to use ALET. An error message is issued otherwise.
4. The ALET operand is supported on both CMS and z/OS, but on z/OS you must create and discover the ALET yourself. As CP uses only the primary list, this flag is added on CMS, that is, ALETS 2 and 01000002 are equivalent. On z/OS the ALET must be specified exactly.
5. Specifying ALET 0 has no effect. You cannot specify ALET 1 on CMS.

strasmfind—Select Statements from an Assembler File as XEDIT Find

strasmfind selects Assembler statements that begin with the specified string. It discards statements that do not begin with the specified string. An Assembler statement can span lines. XEDIT rules for FIND apply.



Type: Selection stage.

Syntax Description: A string is required. The maximum string length is 71 characters.

Operation: Input records are matched the same way XEDIT matches text in a FIND command (tabs 1, image off, case mixed respect):

- A null string matches any record.
- Blank characters in the string represent positions that must be present in the input record, but can have any value.
- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

When the first line of a statement is matched, *strasmfind* copies all lines of the statement without further inspection to the primary output stream, or discards them if the primary output stream is not connected. When the first line of a statement is not matched, *strasmfind* discards all lines of the statement without further inspection, or copies them to the secondary output stream if it is connected.

Input Record Format: An Assembler statement consists of one or more lines. Lines before the last one have a non-blank character in column 72. The last line of a statement is blank in column 72, or shorter than 72 characters.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *strasmfind* strictly does not delay the record.

Commit Level: *strasmfind* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *strasmfind* terminates when it discovers that no output stream is connected.

Converse Operation: *asmnfind*.

See Also: *asmcont*, *asmfind*, and *asmxpnd*.

Examples: To select all statements in an Assembler program that have a label beginning with 'LAB':

```
... | asmfind LAB|...
```

strasmnfind

To select all statements in an Assembler program that have the label 'LAB':

```
... | asmfind LAB_|...
```

The underscore indicates that column 4 must be blank; thus the label is three characters.

To select all comments in an Assembler program:

```
... | asmfind *|...
```

To select all statements of an Assembler program, except comments and those having a label:

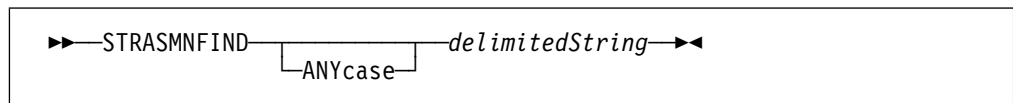
```
... | asmfind _|...
```

Notes:

1. *strasmnfind* does not support changes to the statement format by the ICTL Assembler instruction.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

strasmnfind—Select Statements from an Assembler File as XEDIT NFind

strasmnfind selects Assembler statements that do not begin with the specified string. It discards statements that begin with the specified string. An Assembler statement can span lines. XEDIT rules for NFind apply.



Type: Selection stage.

Syntax Description: A string is required. The maximum string length is 71 characters.

Operation: Input records are matched the same way XEDIT matches text in an NFind command (tabs 1, image off, case mixed respect):

- A null string matches any record.
- Blank characters in the string represent positions that must be present in the input record, but can have any value.
- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

When the first line of a statement is not matched, *strasmnfind* copies all lines of the statement without further inspection to the primary output stream, or discards them if the primary output stream is not connected. When the first line of a statement is matched, *strasmnfind* discards all lines of the statement without further inspection, or copies them to the secondary output stream if it is connected.

Input Record Format: An Assembler statement consists of one or more lines. Lines before the last one have a non-blank character in column 72. The last line of a statement is blank in column 72, or shorter than 72 characters.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *strasmnfind* strictly does not delay the record.

Commit Level: *strasmnfind* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *strasmnfind* terminates when it discovers that no output stream is connected.

Converse Operation: *asmfind*.

See Also: *asmcont* and *asmxpnd*.

Examples: To select labelled or comment statements from an Assembler program:

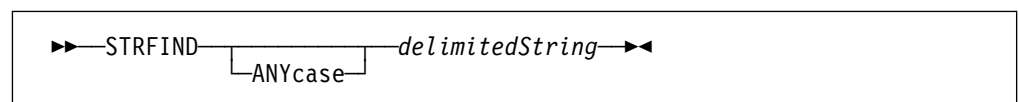
```
...| asmnfind _|...
```

Notes:

1. *strasmnfind* does not support changes to the statement format by the ICTL Assembler instruction.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

strfind—Select Lines by XEDIT Find Logic

strfind selects records that begin with the specified string. It discards records that do not begin with the specified string. XEDIT rules for FIND apply.



Type: Selection stage.

Syntax Description: A keyword is optional. A delimited string is required.

Operation: Input records are matched the same way XEDIT matches text in a FIND command (tabs 1, image off, case mixed respect):

- A null string matches any record.
- Blank characters in the string represent positions that must be present in the input record, but can have any value.
- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

Case is ignored if ANYCASE is specified.

strfind copies records that match to the primary output stream, or discards them if the primary output stream is not connected. It discards records that do not match or copies them to the secondary output stream if it is connected.

strfrlabel

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *strfind* strictly does not delay the record.

Commit Level: *strfind* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *strfind* terminates when it discovers that no output stream is connected.

Converse Operation: *strnfind*.

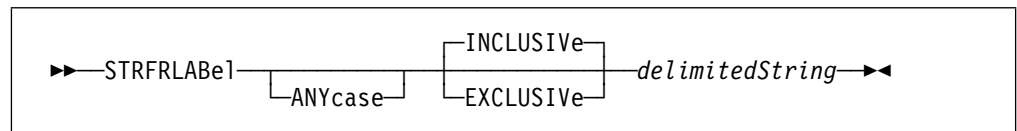
See Also: *find* and *locate*.

Notes:

1. All matching records are selected, not just the first one.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

strfrlabel—Select Records from the First One with Leading String

strfrlabel discards input records up to the first one that begins with the specified string. That record and the records that follow are selected. When EXCLUSIVE is specified, the matching record is also discarded.



Type: Selection stage.

Syntax Description: A string is required.

ANYCASE	Perform caseless compare.
EXCLUSIVE	The matching record is discarded.
INCLUSIVE	The matching record is selected. This is the default.

Operation: Characters at the beginning of each input record are compared with the argument string. When ANYCASE is specified, case is ignored in this comparison. Any record matches a null argument string. A record that is shorter than the argument string does not match.

strfrlabel copies records up to (but not including) the matching one to the secondary output stream, or discards them if the secondary output stream is not connected. It then passes the remaining input records to the primary output stream.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. *strfrlabel* severs the secondary output stream before it shorts the primary input stream to the primary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *strflabel* strictly does not delay the record.

Commit Level: *strflabel* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *strflabel* terminates when it discovers that no output stream is connected.

Converse Operation: *tolabel*.

See Also: *flabel*.

Examples: To discard records on the primary input stream up to the first one beginning with the characters 'abc':

```
/* Skip to first record with label */
'callpipe *: | strflabel /abc/'
```

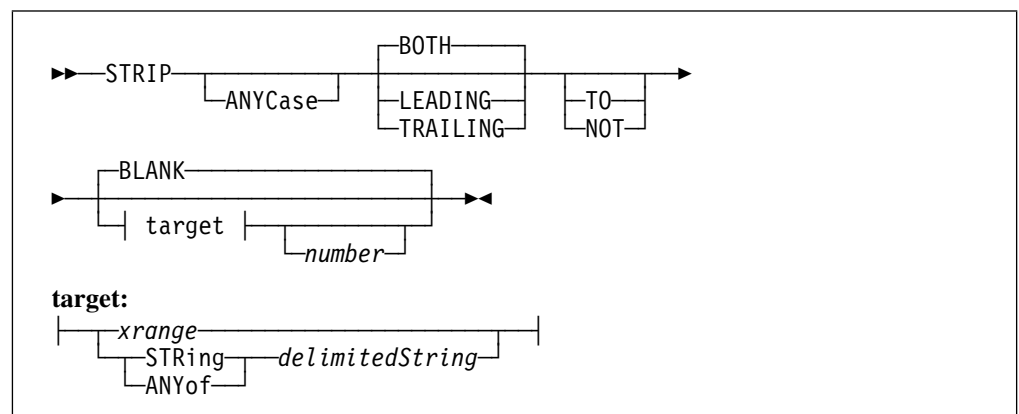
Because this invocation of *strflabel* has no secondary output stream, records before the first one beginning with the string are discarded. The CALLPIPE pipeline command ends when *strflabel* shorts the primary input stream to the unconnected primary output stream; the matching record stays in the pipeline.

Notes:

1. *strfromlabel* is a synonym for *strflabel*.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

strip—Remove Leading or Trailing Characters

strip removes leading or trailing characters or strings.



Type: Filter.

Syntax Description: No arguments are required.

ANYCASE	Ignore case. Conceptually, all processing is done in upper case.
BOTH	Strip from both the beginning and the end of the record.
LEADING	Strip from the beginning of the record.

strliteral

TRAILING	Strip from the end of the record.
NOT TO	Negate the target. Characters not matching the following specification are removed from the record. TO and NOT are synonymous.

The target can be a range of characters or a delimited string. A number is optional after the target. A hex range matches any character within the range. The keyword STRING followed by a delimited string matches the string. The keyword ANYOF followed by a delimited string matches any one character in the string. (The keyword is optional before a one character string, because the effect is the same in either case.) A number after the target limits the number of characters stripped; this can cause part of a string to remain in the record. This number applies independently to each side when stripping BOTH. The default target is a blank; thus, the default is to strip leading and trailing blank characters.

Record Delay: *strip* strictly does not delay the record.

Premature Termination: *strip* terminates when it discovers that its output stream is not connected.

Examples: To remove trailing blanks:

```
pipe literal abc | strip trailing | spec 1-* 1 /*/ next | console
▶ abc*
▶Ready;
```

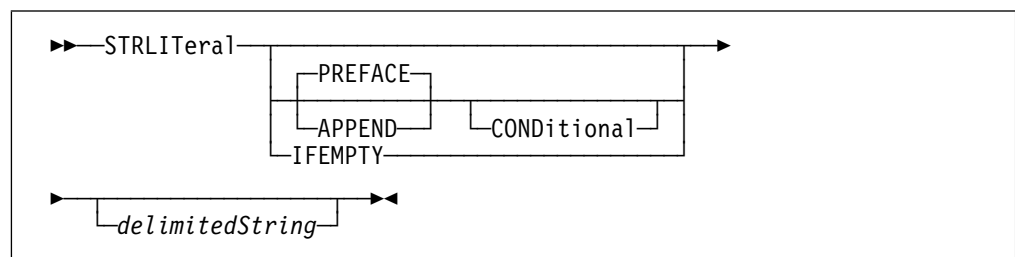
The *specs* stage appends an asterisk to the record to show where it ends.

Notes:

1. The minimum abbreviation of ANYCASE is four characters because ANYOF takes precedence (ANYOF can be abbreviated to three characters).
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

***strliteral*—Write the Argument String**

strliteral writes its argument string into the pipeline before or after it passes records on the input to the output stream.



Type: Device driver.

Syntax Description: One or more keywords are optional to specify if and when a record with the specified string is written to the primary output stream

PREFACE	Write the output record before passing the input to the output.
APPEND	Write the output record after passing the input to the output.
CONDITIONAL	Write the output record only when there is input. That is, if <i>strliteral</i> CONDITIONAL cannot read an input record, it terminates without writing anything.
IFEMPTY	Write the output record only when there are no input records.

Operation: *strliteral* writes a null record when the parameter string is null.

Streams Used: Records are read from the primary input stream and written to the primary output stream. *strliteral* PREFACE shorts the input to the output after it has written the argument string to the pipeline.

Record Delay: The first output record is produced before any input is read. Thus, *strliteral* has the potential to delay one record. *strliteral* with APPEND does not delay the record.

Premature Termination: *strliteral* terminates when it discovers that its output stream is not connected.

See Also: *append*, *literal*, *preface*, and *var*.

Examples: To generate the trailer record in the netdata format:

```
'pipe ... | strliteral append /\INMR06/ | block 80 netdata | punch'
```

To write a heading to the output, but suppress it when there are no output records:

```
pipe cp q v da | locate w4 ,(TEMP), | strliteral cond /T-disks:/ | ...
... console
```

►Ready;

To write a line instead of the output:

```
pipe cp q v da | locate w4 ,(VDSK), | ...
... strliteral ifempty /No V-disk/ | console
```

►DASD 0192 9336 (VDSK) R/W 5000 BLK ON DASD VDSK SUBCHANNEL = 000A

►Ready;

Use two *strliteral* stages, one with CONDITIONAL and one with IFEMPTY, to get a header when there is output and an error message when there is no output.

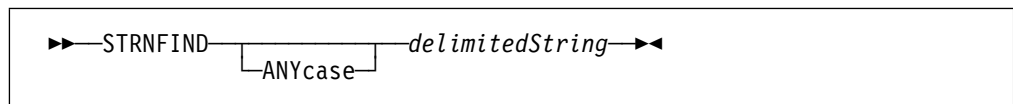
Notes:

1. Records from a cascade of *strliteral* stages appear in the reverse order of their appearance in the pipeline specification unless APPEND is specified; see Figure 62 on page 34.
2. Use *var* to write data that contain stage separators, end characters, and other characters that have a special meaning to the pipeline specification parser.
3. *strliteral* may be used to inject a record in front of the file somewhere downstream in a pipeline, but it can also be a first stage. Note that if you wish to insert a record in front of a file that comes from disk, you must retain the *disk* stage as the first in the pipeline. If not, *disk* appends the single record to the file instead of reading from the file.

4. Be careful when *strliteral* is used where the contents of a stemmed array are being updated or in similar situations where the output overwrites the original data source. Because *strliteral* PREFACE writes the first record before it reads input, this record may be produced before the input has been read; thus, the first record of the updated object may be written before it is read, leading to a “destructive overlap”.
5. *strliteral* IFEMPTY can be useful in front of a *var* stage, to supply a default.
6. It does not make sense to cascade *strliteral* IFEMPTY as the second one will see the first one’s record and thus never write its literal.

strnfind—Select Lines by XEDIT NFind Logic

strnfind selects records that do not begin with the specified string. It discards records that begin with the specified string. XEDIT rules for NFind apply.



Type: Selection stage.

Syntax Description: A string is required. A keyword is optional. A delimited string is required.

Operation: Input records are matched the same way XEDIT matches text in an NFind command (tabs 1, image off, case mixed respect):

- A null string matches any record.
- Blank characters in the string represent positions that must be present in the input record, but can have any value.
- An underscore in the string represents a position where there must be a blank character in the input record.
- All other characters in the string must be equal to the contents of the corresponding position in the input record.

Case is ignored if ANYCASE is specified.

strnfind copies records that do not match to the primary output stream, or discards them if the primary output stream is not connected. It discards records that match or copies them to the secondary output stream if it is connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *strnfind* strictly does not delay the record.

Commit Level: *strnfind* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *strnfind* terminates when it discovers that no output stream is connected.

Converse Operation: *strfind*.

See Also: *nlocate* and *nfind*.

Examples: To discard lines with 'a' in column 1 and 'c' in column 3:

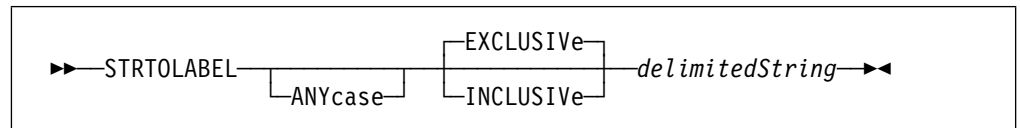
```
pipe literal abc axc axy xyc | split | strnfind /a c/ | console
▶axy
▶xyc
▶Ready;
```

Notes:

1. *notfind* is a synonym for *strnfind*.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

strtolabel—Select Records to the First One with Leading String

strtolabel selects input records up to the first one that begins with the specified string. That record and the records that follow are discarded. When INCLUSIVE is specified, the matching record is also selected.



Type: Selection stage.

Syntax Description: A string is required.

ANYCASE	Perform caseless compare.
EXCLUSIVE	The matching record is discarded. This is the default.
INCLUSIVE	The matching record is selected.

Operation: Characters at the beginning of each input record are compared with the argument string. When ANYCASE is specified, case is ignored in this comparison. Any record matches a null argument string. A record that is shorter than the argument string does not match.

strtolabel copies records up to (but not including) the matching one to the primary output stream, or discards them if the primary output stream is not connected. If the secondary output stream is defined, *strtolabel* then passes the remaining input records to the secondary output stream.

The matching record stays in the pipeline if the secondary output stream is not defined; it can be read again if the current pipeline is defined with CALLPIPE.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. If the secondary output stream is defined, *strtolabel* severs the primary output stream before it passes the remaining input records to the secondary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *strtolabel* strictly does not delay the record.

Commit Level: *strtolabel* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

structure

Premature Termination: *strtolabel* terminates when it discovers that no output stream is connected.

Converse Operation: *strfrlabel*.

See Also: *between*, *inside*, *notinside*, *outside*, *tolabel*, and *whilelabel*.

Examples: To load records up to the first one beginning with `'.end'`:

```
/* Load batch of records into stem */  
'callpipe *: | strtolabel /.end/| stem todo.'
```

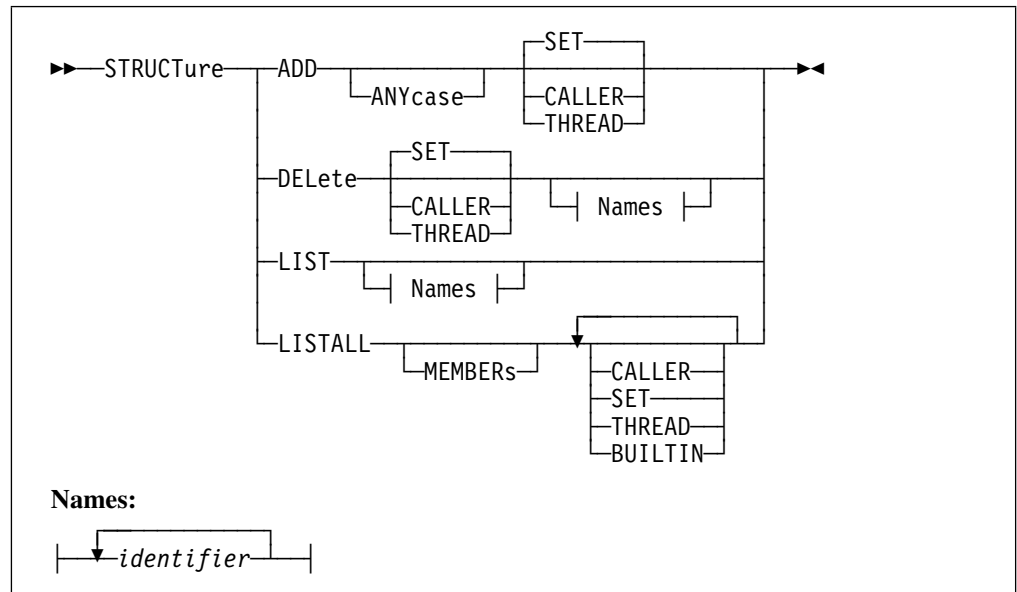
strtolabel is before the *stem* stage that loads the variables; all lines would be processed if the order of the stages were reversed.

Notes:

1. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

structure—Manage Structure Definitions

structure maintains *CMS Pipelines*'s defined structures. Depending on the keyword specified, it defines, deletes, or lists structure definitions.



Type: Gateway.

Placement: *structure* LISTALL must be a first stage.

Syntax Description:

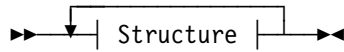
:	ADD	Add to the set of active structures. The structure definitions are read from the primary input stream. The default is to define structures in set scope having case sensitive identifiers. Specifically, DELETE and LIST requests for such a structure will also be caseless.
:		
:		
:		
:		
:	ANYCASE	Structures defined when ANYCASE is active are marked as caseless. That means that case is ignored in all searches for structure names and member names within such a structure, but case may still be respected in embedded structures, depending on how they, in turn, are defined.
:		
:	CALLER	Define structures in caller scope.
:	SET	Define structures in set scope.
:	THREAD	Define structures in thread scope.
:	DELETE	Remove from the set of active structures. Additional names are read from the primary input stream. The default is to remove structures from the topmost set scope.
:		
:	CALLER	Remove structures from caller scope.
:	SET	Remove structures from set scope.
:	THREAD	Remove structures from thread scope.
:	LIST	Write the definition of one or more active structures. Additional names are read from the primary input stream. The specified names are searched in all four structure scopes including nesting caller and set scopes.
:		
:	LISTALL	Write the names of all defined structures in specified scopes. The default is CALLER, SET, and THREAD. The order of listing is the same as the search order: caller, set, thread, and built in. Thus caller and set scope structures are intermixed.
:		
:	MEMBERS	Include the definition of each structure in the list of defined structures.
:		
:	CALLER	List structures in the current caller scope followed by nesting scopes.
:		
:	SET	List structures in the current set scope followed by nesting scopes.
:		
:	THREAD	List structures in thread scope.
:		
:	BUILTIN	List structures that are built in.
:		
:	<i>identifier</i>	The name of a structure. Structure names must begin with a letter in the English alphabet or one of the special characters @#\$!?. The second and subsequent character may also be a digit.
:		

Operation: For *structure* DELETE, the structures specified in the operand list and each set of structures in an input record are processed as a unit. You may delete a structure that embeds other structures in the same scope if all are deleted in the same record. Otherwise

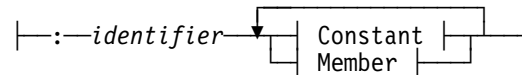
structure

you cannot remove a structure that is embedded in another structure; you must delete the embedding structure first. For caller and set scope, you can delete from the outermost scope only.

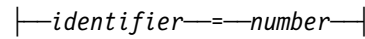
Input Record Format: For *structure* ADD, the input is in free form; it may be spanned across lines. The syntax of the input stream is:



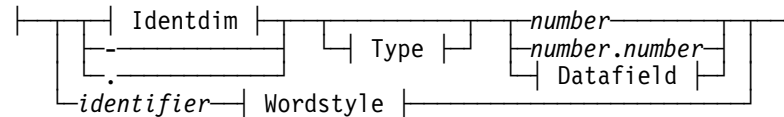
Structure:



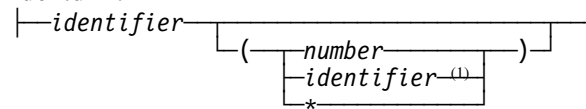
Constant:



Member:



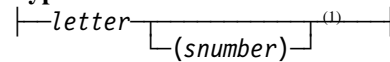
Identdim:



Note:

¹ The identifier must be declared as a manifest constant.

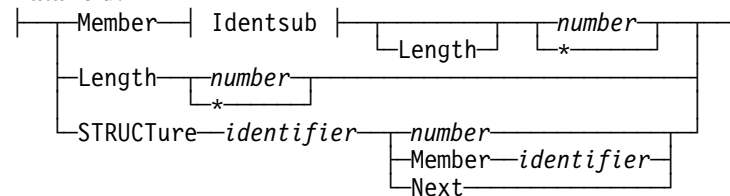
Type:



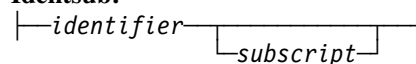
Note:

¹ No blanks are allowed from the letter to the right parenthesis, if a scale is present.

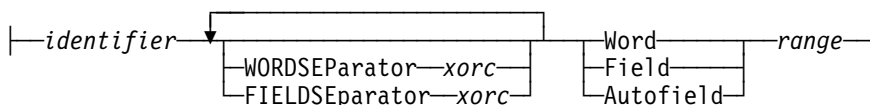
Datafield:



Identsub:



Wordstyle:



The input to *structure* ADD defines one or more structures, or is empty (consists of blanks only). The first non-blank character must be a colon. A colon marks the beginning of the definition of a structure.

For each structure, the first word after the colon specifies the name of the structure.

The structure identifier must not exist within the innermost of the specified scope (there is one thread scope only). In caller or set scope, the structure may also exist in thread scope or in one of the nesting caller or set scopes. In particular, it is allowed to define a structure in set scope that is obscured by an already defined structure in caller scope within the current set; likewise an obscured structure can be defined in thread scope. Thus, the new definition replaces the existing one(s) until it is removed or the scope ends.

Input data up to the next colon or to end-of-file define the contents of the structure. The definition is written as tokens that are delimited by blanks or line ends. The members may be manifest constants or members. During definition, a current position is maintained as the next available position after the last defined member, unless that definition precludes such a definition.

A manifest constant is a symbolic reference to a number; it is specified by an identifier, an equal sign, and a number.

A member defines a range of the record. The first word of the member definition contains an identifier, which must be unique within the structure being defined, optionally followed, in parentheses, by a dimension, which is a positive number, an identifier for a manifest constant, or an asterisk indicating an unbounded array. No position is established when the dimension is an asterisk. Members come in two flavours, which can be intermixed:

- “Proper” members define a fixed number of columns in the record; they can be chained indicating that the member immediately follows the previous one (when a position has been established). A hyphen or period instead of the member identifier specifies an unnamed filler.
- The word style is, in effect, a symbolic name for a range of words or fields. Such members do not establish a position.

The balance of this section describes the first type only.

A single letter other than L defines the type of the member (L is the abbreviation of LENGTH). *structure* makes this type upper case, but it does not attach any particular meaning to it; however, *pick* and *spec* do for types C, D, F, P, R, and U.

Refer to “Using Typed Data” on page 93 for the description of typed data.

A signed number in parentheses is optional after the type character. If present, there must be no blanks in the type and number. The number is restricted to -32768 through 32767. Again, *structure* attaches no meaning to this number, but *pick* and *spec* interpret it as a scale factor when the type is P (packed decimal); that is, a positive number specifies the number of decimal places after the implied decimal point.

structure

Following this single letter and optional number, you specify the location of the member in the record. This can be:

- A single number, or two numbers separated by a period (asterisks are not allowed). The first or only number is the beginning column. The second number is the count of bytes in the field; when omitted, the length is one.
- The keyword `MEMBER`, which can be abbreviated down to one letter, followed by an identifier, an optional subscript, and a count. A subscript is a positive number in parentheses. This defines the member as being overlaid on the already defined member for the specified length. This is equivalent to the `ORG` instruction in Assembler parlance.
- The keyword `LENGTH`, which can be abbreviated down to one letter, followed by a number defining the field length, or an asterisk indicating a field of variable length that extends to the end of the record for an input field; for an output field, an asterisk specifies that the output will have the same length as the input. A position must have been established unless this is the first member of the structure, in which case column 1 is the position. There is no current position after the field when an asterisk is specified; such a field does not contribute to the length of the structure as it is potentially infinite.
- The keyword `STRUCTURE` defines an embedded structure, which must have been defined previously, perhaps earlier in the input stream. When adding structures to thread scope, only other structures in thread scope may be referenced; any structure may be referenced when defining in caller scope, but structures in set scope cannot resolve to structures defined within call scope in the outermost set (though they can resolve any structure defined in a nesting set). The keyword is followed by the identifier of the structure being referenced and the position within the embedding structure (the structure being defined). The length of the member is defined by the length of the structure; you cannot specify it explicitly. The position of the embedded structure is specified by:
 - A number, which is the beginning column.
 - The keyword `MEMBER`, which can be abbreviated down to one letter, followed by an identifier and an optional subscript. This defines the structure as being overlaid on the already defined member.
 - The keyword `NEXT`, which can be abbreviated down to one letter. A position must have been established unless this is the first member of the structure, in which case column 1 is the position.

Output Record Format: *structure* `ADD` and *structure* `DELETE` produce no output.

The output from *structure* `LIST` is in a form that can be passed to *structure* `ADD` if comments are removed, for example, by *chop* `<`. Note, however, that it may not be the form used to define the structure, but the two definitions are equivalent.

The output from *structure* `LISTALL` is unspecified.

Streams Used: Records are read from the primary input stream and written to the primary output stream.

Record Delay: *structure* does not delay the record when listing. `&jphsnmaf`. `ADD` delays the line of the structure definition by at least one record.

Record Delay: *structure* does not delay the record.

Premature Termination: *structure* terminates when it discovers that its output stream is not connected. This can happen only when listing structure definitions.

Examples: To define and list a structure:

```
pipe literal : s mbr 1 len 4|struct add|append struct list s|cons
▶:s          <length 4>
▶ mbr       1.01
▶ len       4.01
▶Ready;
pipe struct listall|cons
▶Ready;
```

The structure is defined in set scope. Note that this defines two members rather than specify a length for the first member (to do that you must specify `mbr 1.4` or `mbr len 4`). The *append* stage ensures that the list is made after all input structures are defined. Structure *s* is discarded when the PIPE command terminates, thus the second pipeline produces no output as there is nothing to list.

A really contorted example of a nested pipeline set and an obscured structure:

```
pipe literal : s m 1|struct add thread
▶Ready;
pipe literal : s mbr 1 len 4|struct add| ...
... strliteral after /(stagesep ?) struct listall members?cons/|
... runpipe|cons

▶Set level 1
▶:s          <length 4>
▶ mbr       1.01
▶ len       4.01
▶Thread
▶:s          <length 1>
▶ m         1.01
▶Ready;
pipe struct del thread s
▶Ready;
```

Notes:

1. These caseless structures are built into *CMS Pipelines*:

EVENTRECORD	Records produced by <i>runpipe</i> EVENTS (Member EVENTREC of FPLOM MACLIB).
FPLASIT	The first eighty bytes of a data space from <i>adrspace</i> CREATE INITIALISE.
FPLSTORBUF	The output record from <i>instore</i> .
VMCMHDR	VMCF interrupt header (see VMCLISTEN).
VMCPARM	VMCF parameter list (see VMCLIENT and VMCDATA).
DIRBUFF	The CMS data area, which is the data returned by DMSGETDI.

2. A structure that is not in the current pipeline set can be obscured by defining a structure of the same name in caller or set scope. A structure in the current set is obscured by one in caller scope. A built-in structure is obscured by a definition in any scope.
3. Embedded structure names are resolved when the structure is defined. Obscuring an embedded structure has no effect on already existing definitions.

structure

4. You can delete only structures in thread scope, in the current pipeline set, or in the calling stage; structures cannot be deleted in nesting pipeline sets or caller scopes within those sets, but they can be obscured.
5. *structure* ADD cannot issue messages that relate to the original input records because it conditions the input stream so that a structure definition does not span record boundaries. Instead, it relates messages to the count of complete structure definitions processed and members processed within the structure being defined.
6. Be sure that structures are defined before they are referenced, because defining a structure implies reading input records, which must happen on commit level 0. Thus, it is likely that any reference to a structure in the same pipeline specification would be a reference to an undefined structure.

This can be resolved in three ways:

- Define the structures in thread scope. The drawback is that this may make them more visible than desired, and possibly obscured by other definitions in a nesting set scope.
- Make the PIPE command run a REXX program that issues CALLPIPE to add the structure definitions to caller or set scope before it issues the “real” pipeline, also with CALLPIPE. This is recommended for production strength code. (Remember that an EXEC can invoke itself as a REXX stage; you do not need an additional file, but the EXEC will then be processed twice by the interpreter.)
- Cascade *structure* ADD and *append* to issue a subroutine pipeline after *structure* ADD has ended. This is a handy way, for example, to list the definition of a structure. Note that the entire pipeline must be the argument to APPEND, or at least the stages that reference the newly defined structure. (Double up the vertical bars or use a different stage separator.)

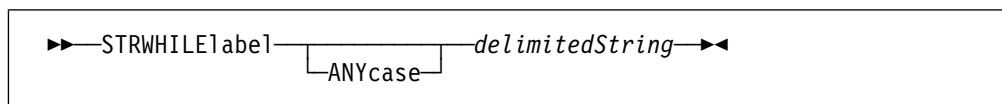
One advantage of the first two approaches is that you can inspect the return code from the pipeline that loads the structures before issuing the “real” pipeline.

7. You may wonder whether it is possible to create a recursion in embedded structures. You can prove by induction that this is not possible because a structure cannot contain itself. This is because the structure is not defined until the next colon or end-of-file is met; nor can it embed an undefined structure. However, a structure definition can embed a structure that it is about to obscure, but that structure could not embed itself when it was defined, so there is still no recursion.
8. The index origin is 1 for arrays. That is, the first member of an array has subscript 1.
9. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
10. *spec* does not allow a question mark in an identifier, as the question mark is parsed as the conditional operator.
11. *structure* starts at commit level 0. This has implications when it is issued with CALLPIPE on a negative commit level, as this commit will force a commit of the caller.
12. The motivation for caller scope is this: Suppose two REXX programs both require a particular structure, perhaps the first creates a record containing the structure and the second formats such a structure, but they do not always run as a cascade, so the second program cannot rely on the first program always defining the structure. Thus each program will wish to add the structure definition, but as it turns out, there is no way for a stage to determine whether it would be successful in adding a structure to the set. It might query by LISTALL, but even when the query indicates that a particular structure does not exist, that does not preclude one from being defined by the time the

- : stage is resumed. Conversely, a REXX stage may define structures in caller scope with
 : impunity.
- : 13. Using ADDPIPE to issue a pipeline specification to define a structure will not increase
 : the commit level of the current pipeline specification, but it is undefined when the
 : structure will be defined and the issuer will be unable to determine whether the
 : definition was successful or not.
- ! 14. The output lines generated for LISTALL and LIST are buffered internally by *structure*
 ! and represent a snapshot of the structures as defined when *structure* processed the
 ! arguments or input record.

strwhilelabel—Select Run of Records with Leading String

strwhilelabel selects input records up to the first one that does not begin with the specified string. That record and the records that follow are discarded.



Type: Selection stage.

Syntax Description: A string is required.

Operation: Characters at the beginning of each input record are compared with the argument string. When ANYCASE is specified, case is ignored in this comparison. Any record matches a null argument string. A record that is shorter than the argument string does not match.

strwhilelabel copies records up to (but not including) the first one that does not match to the primary output stream, or discards them if the primary output stream is not connected. *strwhilelabel* passes the remaining input records to the secondary output stream.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. *strwhilelabel* severs the primary output stream before it passes the remaining input records to the secondary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *strwhilelabel* strictly does not delay the record.

Commit Level: *strwhilelabel* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *strwhilelabel* terminates when it discovers that no output stream is connected.

See Also: *between*, *flabel*, *inside*, *notinside*, *outside*, and *tolabel*.

Notes:

- : 1. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
 ! 2. *pick* can do what *strwhilelabel* does and much more.

stsi—Store System Information

stsi writes a single record containing the information requested.

▶▶—STSI—hexString—◀◀

Type: Arcane device driver.

Placement: stsi must be a first stage.

Syntax Description: The operand consists of three hexadecimal digits that specify the configuration level to write. These configurations can currently be requested: 111, 121, 122, 221, 222, 322.

Output Record Format: The output record is 4096 bytes long, as this is the size of the system information block specified by the architecture, but most of it contains binary zeros. Refer to the *z/Architecture Principles of Operation* for the contents of the system information block.

Premature Termination: stsi terminates when it discovers that its output stream is not connected.

Publications: *z/Architecture Principles of Operation*, SA22-7832

subcom—Issue Commands to a Subcommand Environment

subcom issues commands to a subcommand environment without intercepting terminal output.

▶▶—SUBCOM—word—
└──string──┘—◀◀

Type: Host command interface.

Syntax Description: A word is required. A string is allowed, when no secondary output stream is defined.

Operation: The first blank-delimited word of the argument string is the name of the subcommand environment to process the commands. If there is no environment with the name specified, the environment name is translated to upper case. The remainder of the argument string (if present) and input lines are issued as commands to the subcommand environment.

Input records are passed to the output after the command is issued; no output is produced on the primary output stream for a command specified as operands to subcom.

On z/OS, the default REXX environment is searched for the subcommand environment, even when subcom is in a pipeline specification that was issued from a REXX filter (which runs in a reentrant environment).

When the secondary output stream is defined, the return code is written to this stream after each command has been issued and the command has been written to the primary output stream.

Streams Used: Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *subcom* strictly does not delay the record. When the secondary output stream is defined, the record containing the return code is written after the input record is passed to the output.

Commit Level: *subcom* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: When the secondary output stream is not defined, *subcom* terminates as soon as it receives a negative return code on a command. The corresponding input record is not copied to the output and it is not consumed. When the secondary output stream is defined, *subcom* terminates as soon as it discovers that this stream is not connected. If this is discovered while a record is being written, the corresponding input record is not consumed.

See Also: *aggrc*, *cms*, *command*, *cp*, *starmsg*, *xedit*, and *xmsg*.

Examples: *subcom* is often used to send commands to XEDIT; this example shows how to insert records into the current file:

```
...| change //i / | subcom xedit
```

Remember that the lines are processed according to the XEDIT settings of CASE, IMAGE, and so on.

Use the fact that *subcom* copies input lines to the output after the command has been issued to write a line to the console:

```
/* Append ready message to commands */
'PIPE immcmd CMS',
  '| subcom CMS',      /* This won't "trap" the command output */
  '| spec /Ready!/ 1',
  '| console'
```

Notes:

1. Use *cms* (or *command*) to pass a command on to SUBCOM EXEC if you wish to issue a subcommand and intercept terminal output:

```
/* SUBCOM EXEC: Issue command to a subcommand environment      */
signal on novalue
parse arg where command
address value where
'command
exit RC
```

2. Null and blank input lines are issued to the subcommand environment. The CMS subcommand environment ignores such commands, but this should not be taken as a general rule; it is clearly up to the individual environment to interpret the command string it is issued. Likewise, XEDIT ignores blank and null lines, unless it is in insert mode. XEDIT, for instance, will insert a blank line into the file in input mode.


:
:

- Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

Return Codes: When a secondary output stream is not defined and a negative return code is received on a subcommand, the return code from *subcom* is that negative return code. When a secondary output stream is not defined and the return code is zero or positive, all input records have been processed; the return code is the maximum of the return codes received. When the secondary output stream is defined, the return code is zero unless an error is detected by *subcom*.

substring—Write substring of record

substring writes the specified input range to the output.



Type: Filter.

Syntax Description:

Record Delay: *substring* strictly does not delay the record.

Premature Termination: *substring* terminates when it discovers that its output stream is not connected.


See Also: *spec*.

Notes:

- substring* is an optimisation for a special case of *spec*.
- Use *substring* instead of *not chop number*.

synchronise—Synchronise Records on Multiple Streams

synchronise forces records on parallel streams of a pipeline to move in unison through the pipeline. *synchronise* waits until there is a record available on every input stream and then copies one record from each input stream to the corresponding output stream. It copies no further records to its output until there is again a record available on each input stream. With *synchronise*, the records on one stream can be used to pace the flow through the pipeline of the records on some other stream.



Type: Arcane gateway.

Operation: *synchronise* processes a record from all input streams in this cycle:

- It peeks at each input stream, beginning with the primary input stream and proceeding in numerical order.

- When all input streams have a record available (that is, all streams have been peeked), the records are written to the corresponding output streams, beginning with the primary output stream and proceeding in numerical order.
- Only when all output streams have been written successfully are the input records consumed, beginning with the primary input stream and proceeding in numerical order.

Streams Used: All input streams are read; all output streams are written.

Record Delay: *synchronise* synchronises its input streams. It strictly does not delay the record.

Premature Termination: *synchronise* terminates as soon as it meets end-of-file on any input or output stream.

: That is, when *synchronise* terminates because end-of-file is met on an input stream, no
 : input has been consumed for this set of records. When *synchronise* discovers that an
 : output stream is not connected, a record has been written to streams that have lower
 : numbers.

Examples: *synchronise* can be used to tie the processing of records to external events, such as the receipt of a message from the *MSG system service.

```
/* PACER REXX: Use external events to pace record processing */
'CALLPIPE (endchar ?) *.input: | sync: synchronize | *.output:',
  '? starmsg | sync: | hole'
Exit RC*(RC<>12)
```

synchronise peeks a record from its primary input stream, which is connected to the calling pipeline, but it does not process that record until the *starmsg* stage has captured a message and made it available on the secondary input stream of *synchronise*. *synchronise* then copies the record that it received from the calling pipeline to its primary output stream, which is also connected to the calling pipeline, and copies the message record to its secondary output stream, which is connected to the *hole* stage. Thus, only one record flows through the calling pipeline for each message received from *MSG.

synchronise is useful to control an infinite supply of records going into a stream of *overlay*, for instance to provide a background grid.

```
/* gridit REXX */
'callpipe (end ?)',
  '|literal' copies(left('.', 10), 7), /* Grid line */
  '|dup *', /* Infinite supply of these */
  '|sync: synchronise', /* While there are input lines */
  '|o:overlay', /* Overlay the two */
  '|*.output:', /* Write output */
  '|?*.input:', /* Input file */
  '|sync:', /* Synchronise grid with this stream */
  '|o:' /* Overlay. */
exit RC
```

duplicate * produces as many records as it can, but it cannot produce another record until the previous record has been consumed by *synchronise*. Once *synchronise* has received input on its secondary input stream, which is connected to the calling pipeline, it copies one record from each stream to its corresponding output streams, which are connected to the input streams for *overlay*. *overlay* overlays the record from its secondary input stream on the record from its primary input stream and then writes the combined record on its

primary output stream, which is connected to the calling pipeline. Thus, when the records are returned to the calling pipeline, they have had a background grid added to them. The purpose of using *synchronise* here is to prevent *duplicate* * from flooding the *overlay* stage with input records.

To run a stage (here *udp*) until it produces an output record, store the record in a variable, and then force the stage to terminate because its output stream is severed, without the record being consumed:

```
/* TFTPUDP REXX -- Destroy socket after reading lines from it.      */
signal on novalue
signal on error
do forever
  'callpipe (end \ name TFTPUDP)',
    '|udp 69',                /* Listen on port          */
    '|s:synchronise',        /* Cheat it to get a line  */
    '|stem dgram.',          /* Load into stem         */
    '\literal',              /* Get a null line        */
    '|s:'                     /* And synchronise with   */
If dgram.0=0                 /* Was it forced to stop? */
  Then exit
  'output' dgram.1           /* Write the line         */
end
error: exit RC*(RC<>12)
```

synchronise first waits for *udp* to produce a record. When the record becomes available on its primary input stream, *synchronise* then peeks at the null record on its secondary input stream. It now has a record on all defined input streams, so it writes the record from its primary input stream to its primary output stream, where *stem* stores it in the stemmed array. But when it tries to write the null record from its secondary input stream, it discovers that its secondary output stream is not connected, so it terminates without consuming either input record. This causes *udp* to terminate, because its OUTPUT command receives a return code of 12 (end-of-file).

The point is that *udp* is forced to terminate immediately rather than when it tries to write the next record. That is, the resource used by *udp* is released as soon as it has produced one record, thus immediately becoming available to be used elsewhere.

Notes:

1. *synchronise* has been used in front of *spec* in the past to make *spec* terminate as soon as one of its input streams reached end-of-file. This usage should be replaced with *spec* STOP ANYEOF.

sysdsn—Test whether Data Set Exists

sysdsn calls the REXX function *sysdsn()* and writes the function's result to the output stream.



Type: Device driver.

Syntax Description: If an argument string is specified, it is processed before *sysdsn* reads input records, as if it were an input record.

Record Delay: *sysdsn* strictly does not delay the record.

Commit Level: *sysdsn* starts on commit level -2. It commits to level 0 before processing data.

Premature Termination: *sysdsn* terminates when it discovers that its primary output stream is not connected.

See Also: *listdsi* and *state*.

Examples: To test for the existence of a data set:

```

pipe sysdsn tso.exec | terminal
▶OK
▶READY
pipe sysdsn exec | terminal
▶DATASET NOT FOUND
▶READY
pipe sysdsn ? | terminal
▶INVALID DATASET NAME, ?
▶READY

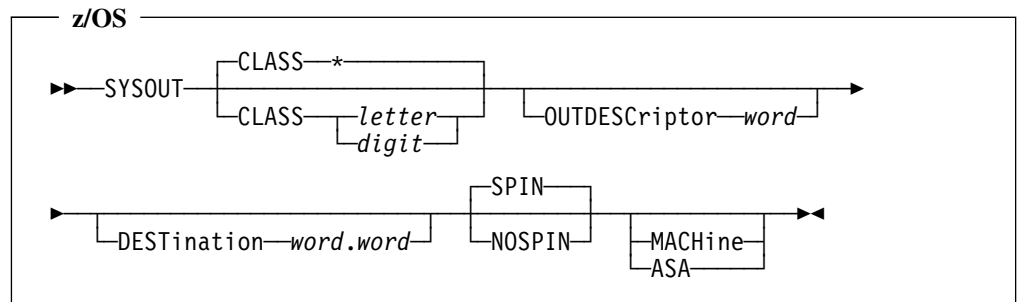
```

Notes:

1. Data set names follow the TSO conventions. Enclose a name that is fully qualified in single quotes. The prefix is applied to data set names that are not enclosed in quotes.

sysout—Write System Output Data Set

sysout writes a system output data set.



Type: Device driver.

Syntax Description:

CLASS Specify the output class. Asterisk, which is the default, selects the default output class for the job. The class can be a letter or a digit; letters are translated to upper case.

OUTDESCRIPT	Specify an output descriptor, which has been defined by the TSO command OUTDES or the JCL statement OUTPUT. The output descriptor can be one to twelve characters; it is translated to upper case. By default, no output descriptor is associated with the data set. Only one output descriptor is supported.
DESTINATION	Specify the destination node and user ID separated by a period. The two words are translated to upper case and truncated after eight characters.
SPIN	Release the data set as soon as it is closed. This is the default.
NOSPIN	Release the data set at the end of the job.
MACHINE	The records contain machine carriage control in the first column.
ASA	The records contain ASA carriage control in the first column.

sysout allocates the data set, opens it, and then commits to level 0.

Operation: *sysout* writes each input record to SPOOL and then copies it to the output, if it is connected.

Record Delay: *sysout* does not delay the record.

Commit Level: *sysout* starts on commit level -2000000000.

See Also: > and >>.

Examples: To send a message in one card:

```
pipe literal Hello, there? | sysout class b dest dkibvm2.john
```

Notes:

1. If the options supported by *sysout* are not adequate for your application, use then ALLOCATE command to allocate a SYSOUT data set and then use > DD= to write the data set.
2. Specify a class on TSO. The default output class is usually purged.
3. *printmc* is a synonym for *sysout*, which sets MACHINE by default.
4. *punch* is a synonym for *sysout*, which does not set carriage control by default.
5. Option code J is not supported.

sysvar—Write System Variables to the Pipeline

sysvar writes the contents of system variables to the pipeline. Specify variable names as arguments, on input records, or both.



Type: Device driver.

Syntax Description: The arguments are optional.

Operation: The contents of the system variables specified in the arguments string (if any) are written to the pipeline. For each input record, the contents of the specified system variables are written to the pipeline.

Input Record Format: System variable names separated by blanks.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *sysvar* does not delay the last record written for an input record. It does not delay the response to an input record that contains a single word. It produces all output records before consuming the input record that contains the corresponding variable names.

Premature Termination: *sysvar* terminates when it discovers that its output stream is not connected. *sysvar* also terminates if an undefined variable is referenced.

See Also: *tso*.

Examples: To display the user identification:

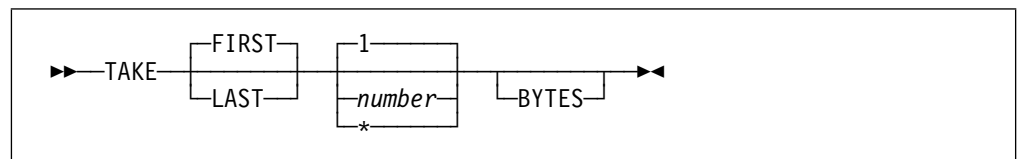
```
pipe sysvar sysuid | console
▶DPJOHN
▶READY
```

Notes:

1. *sysvar* is both a filter and a host command interface. It is classified as a filter because it terminates as soon as its output stream is not connected. Querying variables has no side effects; there is no point in continuing when the result of the query is discarded.

take—Select Records from the Beginning or End of the File

take FIRST selects the first *n* records and discards the remainder. *take* LAST discards records up to the last *n* and selects the last *n* records.



Type: Selection stage.

Syntax Description: The arguments are optional. Specify a keyword, a number, a keyword, or any combination.

FIRST	Records are selected from the beginning of the file. This is the default.
LAST	Records are selected from the end of the file.
<i>number</i>	Specify the count of records or bytes to select. The count may be zero, in which case nothing is selected.
*	All records are selected.
BYTES	The count is bytes rather than records.

tape

Operation: When BYTES is omitted, *take FIRST* copies the specified number of records to the primary output stream, or discards them if the primary output stream is not connected. If the secondary output stream is defined, *take FIRST* then passes the remaining input records to the secondary output stream.

take LAST stores the specified number of records in a buffer. For each subsequent input record (if any), *take LAST* writes the record that has been longest in the buffer to the secondary output stream (or discards it if the secondary output stream is not connected). The input record is then stored in the buffer. At end-of-file *take LAST* flushes the records from the buffer into the primary output stream (or discards them if the primary output stream is not connected).

When BYTES is specified, operation proceeds as described above, but rather than counting records, bytes are counted. Record boundaries are considered to be zero bytes wide. In general, the specified number of bytes will have been taken in the middle of a record, which is then split after the last byte. When FIRST is specified the first part of the split record is selected and the remainder is discarded. When LAST is specified, the first part of the split record is discarded and the second part is selected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. *take FIRST* severs the primary output stream before it shorts the input to the secondary output stream. *take LAST* severs the secondary output stream before it flushes the records from the buffer to the primary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *take FIRST* does not delay the record. When BYTES is not specified *take LAST* delays the specified number of records. When BYTES is specified, *take LAST* delays the number of records to needed for the specified number of bytes.

Commit Level: *take* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *take* terminates when it discovers that no output stream is connected.

Converse Operation: *drop*.

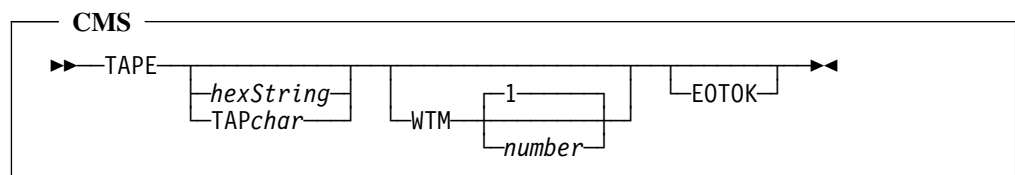
See Also: *flabel* and *tolabel*.

Examples: To select the last line of a command's response:

```
pipe cms query disk | take last | ...
```

tape—Read or Write Tapes

tape connects the pipeline to a tape drive. The tape is read into the pipeline when *tape* is first in a pipeline; records are copied from the pipeline to the tape when *tape* is not first in the pipeline.



Type: Device driver.

Warning: *tape* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *tape* is unintentionally run other than as a first stage. To use *tape* to read data into the pipeline at a position that is not a first stage, specify *tape* as the argument of an *append* or *preface* control. For example, `|append tape ...|` appends the data produced by *tape* to the data on the primary input stream.

Syntax Description: One word is optional when *tape* is first in a pipeline; a word, a keyword with an optional number, and a keyword are optional when *tape* is not first in a pipeline. The first argument is a hexadecimal address, or four characters that are translated to upper case and copied to the CMS RDTAPE or WRTAPE parameter list without inspection. TAP0 through TAPF are the only practical specifications. Refer to *z/VM CMS Macros and Functions Reference*, SC24-6262 for the supported address and names.

When writing to the tape, use the keyword WTM to write one or more tape marks at end-of-file or end of tape; the default is not to write a tape mark. Specify EOTOK to suppress the message issued when the tape is full.

Operation: *tape* reads and writes the tape without positioning it (for instance, the tape is not rewound).

tape writes records to the tape when it is not first in a pipeline. It stops at end-of-file on the input or when the tape drive signals end of tape. Having written the file, *tape* writes tape marks, as requested by the keyword WTM. Message 291 is issued at end of tape after the tape marks (if any) are written, unless suppressed with the keyword EOTOK. A control stage can invoke *tape* repetitively to write a multivolume file from a single input file.

tape does not inspect tape labels (you can create tape labels with *CMS Pipelines* if you wish). Use WTM to write a tape mark after a file is written to tape. Other tape control operations are performed with the CMS TAPE command or equivalent.

tape handles blocks up to 65535 characters (64K-1), which is the maximum length for the underlying CMS interface.

Streams Used: *tape* passes the input to the output.

Record Delay: *tape* strictly does not delay the record.

Premature Termination: When it is first in a pipeline, *tape* terminates when it discovers that its output stream is not connected. When it is writing to a tape, *tape* terminates when the end of tape indicator is turned on.

See Also: *qsam*.

Examples: To copy one tape to another one:

tape

```
/* Tape copy */
address command
signal on error
'TAPE REW'
'TAPE REW (TAP2'
'TAPE MODESET (TAP2 DEN 6250'
do until recs.1=0
  'PIPE tape | tape TAP2 wtm | count lines | stem recs.'
end
error: exit RC
```

To write a file on several unlabelled volumes, switching repeatedly between tapes 181 and 182:

```
/* Multivolume unlabelled tape write */
signal on novalue
tapes='181 182'
do forever
  parse var tapes tape tapes /* Get drive to use */
  tapes=tapes tape /* Put it at the end of the list */
  'callpipe (name MVULTAPE)',
    '|*:', /* Input file */
    '|tape' tape 'wtm eotok' /* Write tape */
  If RC/=0
    Then exit RC
  'peekto' /* End-of-file? */
  If RC/=0
    Then leave /* Most likely */
  address command 'TAPE RUN (' tape
  'callpipe cp MSG OPERATOR Please mount next tape.'
end
exit RC*(RC-=12)
```

To write a trailer file:

```
/* Write EOT or EOF trailer records */
signal on error
do forever
  'callpipe *: | tape wtm eotok | count lines | var blocks'
  signal off error
  'peekto'
  signal on error
  if RC=12 then leave /* Done */
  'callpipe literal EOT1' blocks'| tape wtm eotok'
end
'callpipe literal EOF1' blocks'| tape wtm eotok'
error: exit RC
```

To extract the data records from a file in CMS TAPE DUMP format that contains records of variable length:

```
/* Read dumped file */
'PIPE',
  ' tape',
  '|whilelabel' '02'x || 'CMSV' ||,
  '|spec 6-*',
  '|deblock cms',
  '|> tape file a'
```


Notes:

1. A tape mark is written on output only if you ask for it. This lets you build a tape file with multiple pipeline commands. Remember to write a tape mark when you want one.
2. CMS tapes usually end with two tape marks.
3. *tape* does not convert to and from the TAPE DUMP format; it reads and writes blocks from a tape.
4. Some (if not all) tape units specify that certain command sequences are not valid, but do not check for such sequences. Refer to the reference manual for the tape drive you are using when mixing reads and writes to a tape. Usually it is required that a tape mark be spaced over before starting to write after it. Therefore, if you have been reading a file and wish to write after the tape mark that terminated the read operation, you must perform a backward space file (TAPE BSF) followed by a forward space file (TAPE FSF) before invoking the pipeline to write to the tape.
5. Many tape drives require a minimum of 18 bytes in a block. Shorter blocks may be considered noise.

: tcpcksum—Compute One’s complement Checksum of a Message

: *tcpcksum* computes the one’s complement checksum of the input record and optionally
 : stores it at a specified location in the record.



Type: Filter.

Operation: Without an operand, *tcpcksum* computes the checksum of each input record and produces a 16-bit result checksum on its output. This result is all zeros when the input message contains a valid TCP/IP checksum field.

When specified, the operand designates the begin column of the 16-bit checksum field within the record. The checksum of the record is computed and stored into the specified position; the updated record is then written to the output. For correct interoperability with TCP/IP, the checksum field in the input record must contain binary zeros.

Record Delay: *tcpcksum* does not delay the record.

Premature Termination: *tcpcksum* terminates when it discovers that its output stream is not connected.

See Also: *crc*.

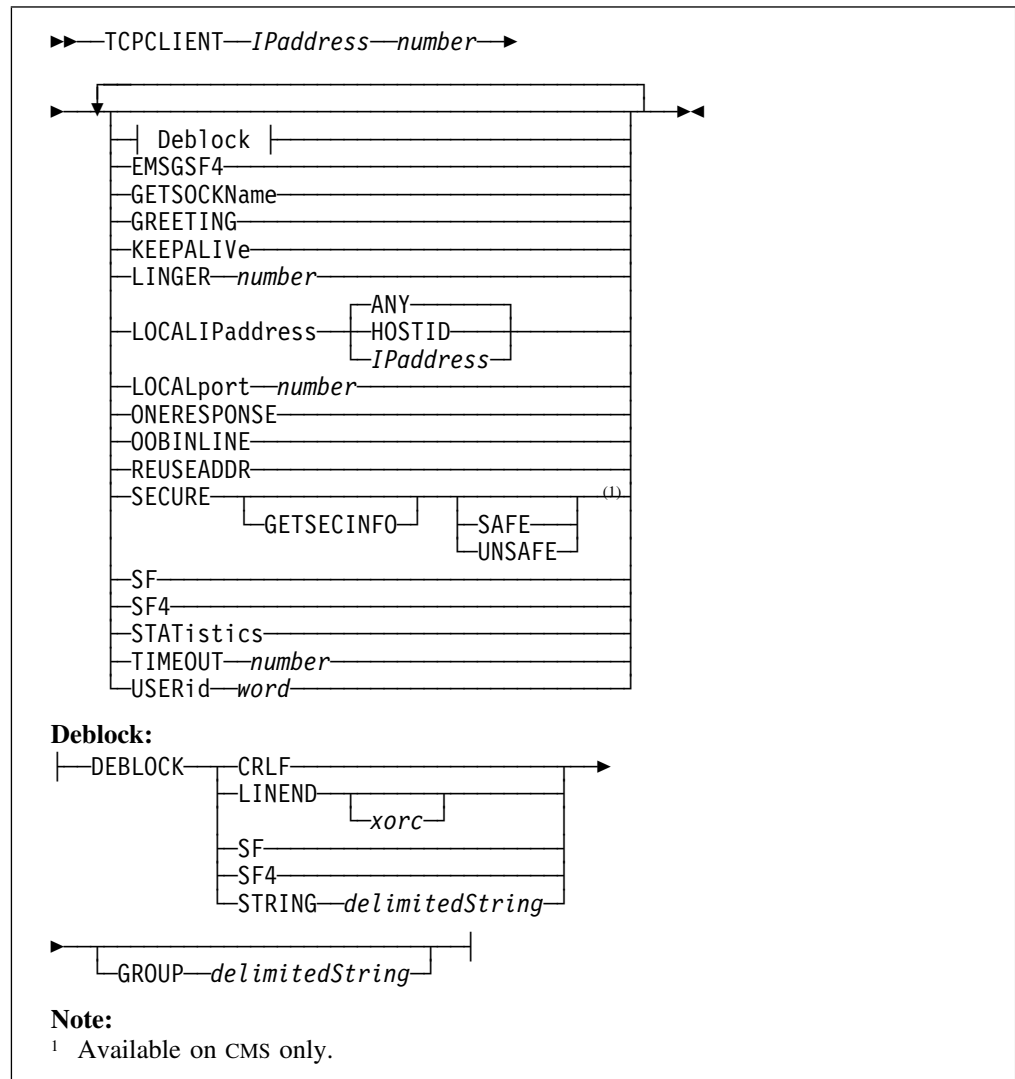
Notes:

1. *tcpcksum* interoperates with the IP, TCP, and UDP headers.
2. Refer to RFCs 1071, 1141, 1624, and 1936.

tcpclient

tcpclient—Connect to a TCP/IP Server and Exchange Data

tcpclient connects itself to a TCP/IP server using the Transmission Control Protocol, optionally secured by z/VM System SSL. It transmits its input records to the server and writes data it receives from the server onto its output stream.



Type: Device driver.

Syntax Description: Two positional operands are required. The first operand specifies the address of the host where the server is running. The address is specified as a “dotted-decimal” number (for example, 9.55.5.13) or (on CMS) as a host name or a host name and a domain (for example, jph.dk.ibm.com). The second operand specifies the port at which the server is listening.

DEBLOCK	Specify the deblocking to be performed on data received from the socket.
CRLF	The byte stream received is split at the two-byte sequence of carriage return and line feed in EBCDIC (X'0D25'). Lines are considered to be terminated by this sequence.
LINEND	The byte stream received is split at EBCDIC line end characters, which by default are X'25'. Specify a single character or a two-character hexadecimal value. (0A is a "good" value, as this is an ASCII line feed.) Lines are considered to be terminated by a line end character.
SF	Each record received has a two-byte length prefix. The length is a binary number in network byte order; it includes the length of this record descriptor word.
SF4	Data received from TCP/IP is considered to contain records prefixed four-byte length prefix. The length is a binary number in network byte order; it includes the length of this record descriptor word.
STRING	Specify a delimited string for the record delimiter. Lines are considered to be terminated by this string. (0D0A is a "good" string, as this is the ASCII representation of carriage return and line feed.)
GROUP	Specify a stage that will group all responses for a transaction into a single record, or will delete all but the last record of the response. You can specify any stage in the delimited string, but it is useful only if it does not delay the record. The group stage is applied after the deblocking. The stage specified in the delimited string can be a REXX program; it cannot be a cascade of stages.
:	EMSGSF4 Process data with four-byte record descriptors as done by SF4. In addition, the first byte received is inspected for being zero in its five leftmost bits. If these are not all zero, it is assumed that the data received are error messages in ASCII from the process setting up the server (for example, <i>inetd</i> on a UNIX system). If this is the case, message 1287 is issued and all data received are converted to EBCDIC, deblocked and issued as message 39.
GETSOCKNAME	Write the contents of the socket address structure to the primary output stream after the socket is connected.
GREETING	The server is expected to send a single line of response as soon as the client is connected (that is, before it receives any data). Be sure to specify a deblocking option so that <i>tcpclient</i> can determine when the complete line has been received.
KEEPALIVE	Turn on the KEEPALIVE socket option.
LINGER	Specify the number of seconds that <i>tcpclient</i> should wait after it receives end-of-file on its input before it closes the connection. The default is zero. This allows time for a final response to travel across the network.

LOCALIPADDR	Specify the local IP address to be used when binding the socket. The default, ANY, specifies that TCP/IP may use any interface address. (An IP address of binary zeros is used to bind the socket.) HOSTID specifies that TCP/IP should use the IP address that corresponds to the host name. Specify the dotted-decimal notation or (on CMS) the host name for a particular interface to be used.
LOCALPORT	Specify the local port to be bound to the client. The default is zero, which causes TCP/IP to assign the port number. Use this option if a port is reserved for your use.
ONERESPONSE	Expect one response record to each transmitted record.
OOBINLINE	Turn on the OOBINLINE socket option.
REUSEADDR	Turn on the REUSEADDR socket option.
SF	Add a two-byte length field to records being sent. The length field includes its own length; the null record would be transmitted as X'0002'. Expect a two-byte length field in messages received; deblock or block messages and write output records for each complete logical record.
SF4	Add a four-byte length field to records being sent. The length field includes its own length; the null record would be transmitted as X'00000004'. Expect a four-byte length field in messages received; deblock or block messages and write output records for each complete logical record.
SECURE	Indicates that a secure TCP/IP connection is established through z/VM System SSL.
GETSECINFO	Write a record to primary output of <i>tcpclient</i> after completion of the SSL/TLS handshake. The record contains diagnostic information about the secure connection; the contents of the record is unspecified.
SAFE	Request z/VM System SSL to verify that the destination specified as the argument on <i>tcpclient</i> matches the identity of the server stated in the server certificate. This is the default when the destination is specified as host name or host name with domain.
UNSAFE	Does not request z/VM System SSL to verify that the destination specified as the argument on <i>tcpclient</i> matches the identity of the server stated in the server certificate. This is the default when the destination is specified as a dotted-decimal IP address.
STATISTICS STATS	Write messages containing statistics when <i>tcpclient</i> terminates. The format of these statistics is undefined. STATS is a synonym.
TIMEOUT	Specify the timeout value in seconds. <i>tcpclient</i> will terminate after the timeout if it receives no response to sending a transaction. When SF, SF4 or DEBLOCK is specified, <i>tcpclient</i> will read the entire response record; it will time out if any segment does not arrive. When none of these options is specified, <i>tcpclient</i> can ensure only that the first segment of the response arrives within the specified time limit.
USERID	Specify the user ID of the virtual machine or started task where TCP/IP runs. The default is TCPIP.

Operation: Input records are written to the socket as they arrive; records are read from the socket and passed to the primary output stream as they arrive. When SF or SF4 is specified (without specifying DEBLOCK), a record descriptor word is transmitted in front of each input record.

Data received on the socket are written to the primary output stream. When DEBLOCK is specified, the appropriate deblocking stage is inserted into the output stream. When GROUP is further specified, the grouping stage is inserted into the output stream. A response is deemed to have been received only when a record is passed to the stage initially connected to the output of *tcpclient*. Thus, ONERESPONSE and TIMEOUT apply to the point after the deblocking and grouping stages.

When SECURE is specified, *tcpclient* uses z/VM System SSL to initiate the SSL/TLS handshake and secure the TCP/IP connection. When the secondary input stream is connected, the SSL/TLS handshake is initiated when a record becomes available on the secondary input stream; when no secondary input stream is connected, the SSL/TLS handshake is initiated immediately after the TCP/IP connection setup is complete. When GETSECINFO is specified, *tcpclient* writes a record to the primary output stream showing additional information about the SSL/TLS handshake. The record is written when the SSL/TLS handshake is complete.

The primary output stream is severed when end-of-file is received from the socket. The socket shutdown for write function is performed when *tcpclient* discovers that the primary input stream has been severed. If ONERESPONSE is specified, the socket is then closed.

When end-of-file is received on the input stream and LINGER is specified, *tcpclient* waits until the connection is closed or the number of seconds specified has expired, whichever occurs first. No indication is provided as to which event occurs; indeed, they could occur simultaneously.

Streams Used: Secondary streams may be defined. When the secondary output stream is connected, a record is written to it when *tcpclient* terminates after TCP/IP has reported an "ERRNO".

Commit Level: *tcpclient* starts on commit level -10. It connects to the server's port, and then commits to level 0.

Premature Termination: When it is first in a pipeline, *tcpclient* terminates when it discovers that its output stream is not connected.

tcpclient also terminates when an error is reflected by TCP/IP (known as an ERRNO). How it terminates depends on whether the secondary output stream is defined or not.

When the secondary output stream is not connected, error messages are issued to describe the error and *tcpclient* terminates with a nonzero return code.

When the secondary output stream is connected, a single record is written to the secondary output stream; *tcpclient* then terminates with the return code zero. The record written contains the error number; the second word contains the symbolic name of the error number if the error number is recognised by *CMS Pipelines*. The assumption is that a REXX program will inspect the error number and decide whether it should retry the operation, discard the current transaction and retry, or give up entirely.

tcpclient also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *tcpdata*, *tcplisten*, and *udp*.

Examples:

```
pipe literal HELLO | tcpclient 9.55.5.13 7 linger 5 | console
```

Notes:

1. TCP/IP transports a byte stream; you cannot expect record boundaries to be preserved across the network. Use the option SF or SF4 to add record descriptors to the data sent. This presumes that the server expects such record descriptors; this is not the TCP/IP tradition.
2. Many servers expect ASCII commands that are terminated by line ends.
3. *tcpclient* does not perform name resolution on TSO; you must specify the dotted-decimal notation for the location of the server.

On CMS, you can specify a host name or a host name followed by a domain. *CMS Pipelines* calls RXSOCKET to do the actual name resolution. As a consequence, the name is resolved using RXSOCKET rules. This implies that the file TCPIP DATA must be available and must point to the name server. RXSOCKET (unlike *CMS Pipelines*) uses the server virtual machine specified in TCPIP DATA.

4. The LINGER option does not enable the SO_LINGER socket option.
5. *CMS Pipelines* defines error numbers in the 5000 range in addition to the ones defined by TCP/IP:

5000 (EpipeResponseTimedOut) No response was received within the interval specified by TIMEOUT.

5001 (EpipeStopped) The pipeline was signalled to stop by passing a record to *pipestop* or through a similar action.

5002 (EpipeSocketClosed) ONERESPONSE is specified and the socket was closed by the communications partner without it sending a response to a transaction.

CMS Pipelines also defines this error number:

0000 (OKSocketClosed) The connection was closed by the communications partner. The stage is not expecting a response; that is, ONERESPONSE is omitted.

When SECURE is specified, protocol failures detected by the z/VM SSL Server are reported by error number as well. For protocol errors detected in the Cryptographic Services Library, the SSL function return code incremented with 10,000 is reported by *tcpclient*. The SSL function return codes are documented in z/OS System SSL Programming, SC14-7495. You are most likely to encounter these:

1001 (EIBMBADPARM) The z/VM SSL Server may not have the service applied to support host name validation that *tcpclient* requests when the destination is specified as host name. The UNSAFE option can be used to disable host name validation.

1012 (EibmNoTLS) No z/VM SSL Server associated with the VM TCP/IP stack.

4008 (ValidationFailed) The address of the TCP/IP host specified as argument for *tcpclient* does not match the public certificate presented by the host. Verify that the correct host name and domain is specified for which the server certificate was issued. When you have verified the destination and are unable to use the host name that matches the certificate, the UNSAFE option can be used to disable the validation.

10008 (GskCertValidation) The public certificate presented by the remote server could not be verified against a root certificate in the database. Obtain the corresponding root certificate and have it imported in the z/VM System SSL certificate database.

10401 (GskExpired) The certificate has expired or is not yet valid.

10402 (GskNoCiphers) The client and server did not find a common cipher to use for the connection.

10417 (GskSelfSigned) The remote side presented a self-signed server certificate. This is not supported by the z/VM SSL Server.

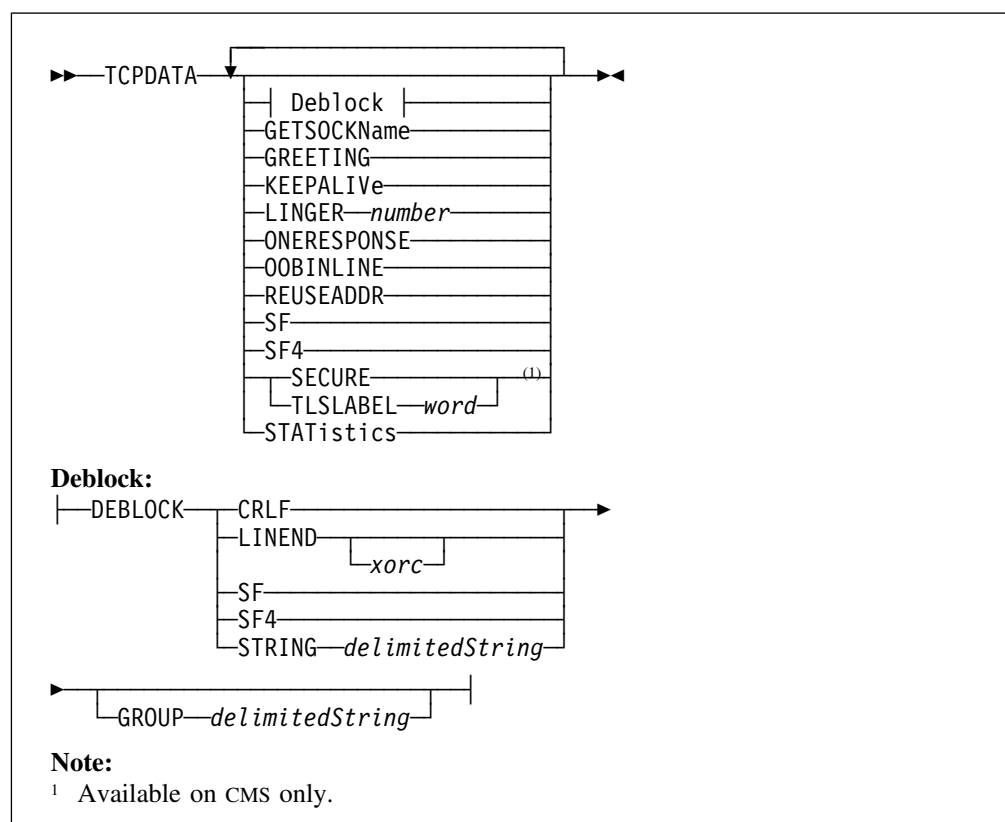
10420 (GskSocketClosed) The remote side closed the connection during the handshake.

The z/VM SSL Server logging may provide additional information that can be helpful to diagnose connection problems.

Return Codes: When the secondary output stream is defined and *tcpclient* terminates due to an error that is reported by TCP/IP as an ERRNO, *tcpclient* sets return code 0 because the error information is available in the record that is written to the secondary output stream. When *tcpclient* terminates because of some other error (for example, if it could not connect to the TCP/IP address space), the secondary output stream is ignored and the return code is not zero, reflecting the number of the message issued to describe this error condition.

tcpdata—Read from and Write to a TCP/IP Socket

tcpdata is used as the stage in a server that receives data from the client and transmits data to the client. The socket to use is described by the first input record, which should have been written by a *tcplisten* stage.



Type: Device driver.

Syntax Description: All operands are optional.

DEBLOCK	Specify the deblocking to be performed on data received from the socket.
CRLF	The byte stream received is split at the two-byte sequence of carriage return and line feed in EBCDIC (X'0D25'). Lines are considered to be terminated by this sequence.
LINEND	The byte stream received is split at EBCDIC line end characters, which by default are X'25'. Specify a single character or a two-character hexadecimal value. (0A is a "good" value, as this is an ASCII line feed.) Lines are considered to be terminated by a line end character.
SF	Each record received has a two-byte length prefix. The length is a binary number in network byte order; it includes the length of this record descriptor word.
SF4	Data received from TCP/IP is considered to contain records prefixed four-byte length prefix. The length is a binary number in network byte order; it includes the length of this record descriptor word.
STRING	Specify a delimited string for the record delimiter. Lines are considered to be terminated by this string. (0D0A is a "good" string, as this is the ASCII representation of carriage return and line feed.)
GROUP	Specify a stage that will group all responses for a transaction into a single record, or will delete all but the last record of the response. You can specify any stage in the delimited string, but it is useful only if it does not delay the record. The group stage is applied after the deblocking. The stage specified in the delimited string can be a REXX program; it cannot be a cascade of stages.
GETSOCKNAME	Write the contents of the socket address structure to the primary output stream after the socket is taken.
:	
GREETING	The client is expected to send an initial message line (that is, before it expects to receive any data). Be sure to specify a deblocking option so that <i>tcpdata</i> can determine when the complete line has been received.
.	
.	
KEEPALIVE	Turn on the KEEPALIVE socket option.
LINGER	Specify the number of seconds that <i>tcpdata</i> should wait after it receives end-of-file on its input before it closes the connection. The default is zero.
ONERESPONSE	Expect one response record to each transmitted record.
OOBINLINE	Turn on the OOBINLINE socket option.
REUSEADDR	Turn on the REUSEADDR socket option.

SF	Add a two-byte length field to records being sent. The length field includes its own length; the null record would be transmitted as X'0002'. Expect a two-byte length field in messages received; deblock or block messages and write output records for each complete logical record.
SF4	Add a four-byte length field to records being sent. The length field includes its own length; the null record would be transmitted as X'00000004'. Expect a four-byte length field in messages received; deblock or block messages and write output records for each complete logical record.
SECURE	Indicates that a secure TCP/IP connection is established through z/VM System SSL. SECURE is mutually exclusive with TLSLABEL.
TLSLABEL	Specifies the label of the certificate defined in the z/VM System SSL certificate database to be used for a secure connection. TLSLABEL is mutually exclusive with SECURE.
STATISTICS STATS	Write messages containing statistics when <i>tcpdata</i> terminates. The format of these statistics is undefined. STATS is a synonym.

Operation: *tcpdata* peeks at the first input record, which contains the information required to take the socket that represents the conversation with the client. When *tcpdata* has obtained the socket, it passes input records to the client and writes data it reads from the socket to the output stream. When SF or SF4 is specified (without specifying DEBLOCK), a record descriptor word is transmitted in front of each input record.

Data received on the socket are written to the primary output stream. When DEBLOCK is specified, the appropriate deblocking stage is inserted into the output stream. When GROUP is further specified, the grouping stage is inserted into the output stream. A response is deemed to have been received only when a record is passed to the stage initially connected to the output of *tcpdata*. Thus, ONERESPONSE and TIMEOUT apply to the point after the deblocking and grouping stages.

When SECURE is specified, *tcpdata* uses z/VM System SSL to initiate the SSL/TLS handshake and secure the TCP/IP connection. When the secondary input stream is connected, the SSL/TLS handshake is expected to be initiated when a record becomes available on the secondary input stream; when no secondary input stream is connected, the SSL/TLS handshake is initiated immediately after the TCP/IP connection setup is complete.

The primary output stream is severed when end-of-file is received from the socket. The socket shutdown for write function is performed when *tcpdata* discovers that the primary input stream has been severed. If ONERESPONSE is specified, the socket is then closed.

Input Record Format: The first record must be in the format written by *tcplisten*:

Pos	Len	Description
1	8	Check word. The constant pipetcp (one trailing blank).
9	8	The ID of the virtual machine or started task that runs TCP/IP. (The USERID operand on the <i>tcplisten</i> stage.)
17	40	The clientid structure filled with information about the <i>tcpdata</i> stage.
57	4	The socket number, binary.

tcpdata

Pos	Len	Description
61	4	A fullword of zeros (for the socket number in takesocket).
65	16	The network address of the client (the structure sockaddr_in).
81	4	The address of <i>tcpdata</i> 's work area.

Streams Used: Two streams must be defined. When the secondary output stream is connected, a record is written to it when *tcpdata* terminates after TCP/IP has reported an "ERRNO".

Commit Level: *tcpdata* starts on commit level -10. It and then commits to level 0.

Premature Termination: *tcpdata* terminates when it discovers that no output stream is connected.

tcpdata also terminates when an error is reflected by TCP/IP (known as an ERRNO). How it terminates depends on whether the secondary output stream is defined or not.

When the secondary output stream is not connected, error messages are issued to describe the error and *tcpdata* terminates with a nonzero return code.

When the secondary output stream is connected, a single record is written to the secondary output stream; *tcpdata* then terminates with the return code zero. The record written contains the error number; the second word contains the symbolic name of the error number if the error number is recognised by *CMS Pipelines*. The assumption is that a REXX program will inspect the error number and decide whether it should retry the operation, discard the current transaction and retry, or give up entirely.

tcpdata also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *tcpclient*, *tcplisten*, and *udp*.

Examples: A simplistic echo server:

```
'callpipe *:| i: fanin | tcpdata | elastic | i:'
```

The input record is fed through *fanin* to *tcpdata* before *fanin* completes the loop that will transmit the response back to the client. The record received is sent unmodified.

Notes:

1. Normally, the server will send a response that is based on a transaction from the client. To do this, the pipeline must have feedback. Be sure to avoid stalls resulting from this; *elastic* is recommended to buffer sufficient records to prevent the stall.
2. TCP/IP may segment transmissions so that *tcpdata* may write a different number of output records (fewer or more, in general) than the corresponding *tcpclient* stage read on its input when it transmitted the data.

If both the server and the client are implemented using *CMS Pipelines*, you can use the SF or SF4 options in both device drivers to maintain the record structure across the byte streams of the network.

If you are writing a server, you can specify that the package as transmitted contains the package length in the first two or four bytes and then use the appropriate option to

simplify your server; but beware that this may not be popular with the client implementers.

3. *CMS Pipelines* defines error numbers in the 5000 range in addition to the ones defined by TCP/IP:

5000 (EpipeResponseTimedOut) No response was received within the interval specified by TIMEOUT.

5001 (EpipeStopped) The pipeline was signalled to stop by passing a record to *pipestop* or through a similar action.

5002 (EpipeSocketClosed) ONERESPONSE is specified and the socket was closed by the communications partner without it sending a response to a transaction.

CMS Pipelines also defines this error number:

0000 (OKSocketClosed) The connection was closed by the communications partner. The stage is not expecting a response; that is, ONERESPONSE is omitted.

When SECURE or TLSLABEL is specified, protocol failures detected by the z/VM SSL Server are reported by error number as well. For protocol errors detected in the Cryptographic Services Library, the SSL function return code incremented with 10,000 is reported by *tcpdata*. The SSL function return codes are documented in z/OS System SSL Programming, SC14-7495. You are most likely to encounter these:

1001 (EIBMBADPARM) The z/VM SSL Server may not have the service applied to support host name validation that *tcpdata* requests when the destination is specified as host name. The UNSAFE option can be used to disable host name validation.

1012 (EibmNoTLS) No z/VM SSL Server associated with the VM TCP/IP stack.

1016 (EibmLabNr) No key is found in the database with the requested label.

10006 (KeyLabelNotFound) When TLSLABEL is used, no key is associated with the label specified. When SECURE is used, no default key is set in the database.

4008 (ValidationFailed) The address of the TCP/IP host specified as argument for *tcpdata* does not match the public certificate presented by the host. Verify that the correct host name and domain is specified for which the server certificate was issued. When you have verified the destination and are unable to use the host name that matches the certificate, the UNSAFE option can be used to disable the validation.

10008 (GskCertValidation) The public certificate presented by the remote server could not be verified against a root certificate in the database. Obtain the corresponding root certificate and have it imported it in the z/VM System SSL certificate database.

10401 (GskExpired) The certificate has expired or is not yet valid.

10402 (GskNoCiphers) The client and server did not find a common cipher to use for the connection.

10417 (GskSelfSigned) The remote side presented a self-signed server certificate. This is not supported by the z/VM SSL Server.

10420 (GskSocketClosed) The remote side closed the connection during the handshake.

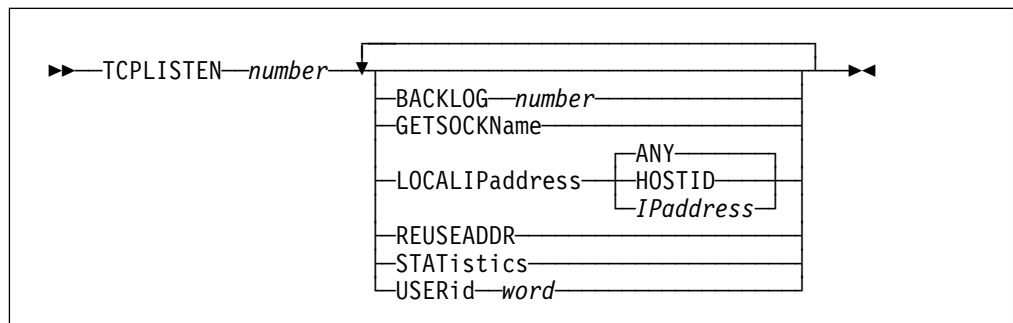
The z/VM SSL Server logging may provide additional information that can be helpful to diagnose connection problems.

4. The LINGER option does not enable the SO_LINGER socket option.

Return Codes: When the secondary output stream is defined and *tcpdata* terminates due to an error that is reported by TCP/IP as an ERRNO, *tcpdata* sets return code 0 because the error information is available in the record that is written to the secondary output stream. When *tcpdata* terminates because of some other error (for example, if it could not connect to the TCP/IP address space), the secondary output stream is ignored and the return code is not zero, reflecting the number of the message issued to describe this error condition.

tcplisten—Listen on a TCP Port

tcplisten listens for and accepts connection requests on a TCP port using the socket interface to TCP/IP and writes a record describing the connection to its primary output stream. The socket representing the connection should be transferred by passing this record to a *tcpdata* stage, which exchanges data with the client. See the operations section below for important usage information.



Type: Device driver.

Syntax Description: Specify as the first operand the number of the port that *tcplisten* should listen on. Specify 0 to have TCP/IP assign the port number; use the GETSOCKNAME option to discover the port number assigned by TCP/IP.

- | | |
|---------------------|--|
| BACKLOG | Specify the maximum number of pending connection requests for the port. The default is 10. |
| GETSOCKNAME | Write the contents of the socket address structure to the primary output stream after the socket is bound. That is, as the first record after <i>tcplisten</i> has committed to level 0, but before it starts listening. |
| LOCALIPADDR | Specify the local IP address to be used when binding the socket. The default, ANY, specifies that TCP/IP may use any interface address. (An IP address of binary zeros is used to bind the socket.) HOSTID specifies that TCP/IP should use the IP address that corresponds to the host name. Specify the dotted-decimal notation or (on CMS) the host name for a particular interface to be used. |
| REUSEADDR | Turn on the REUSEADDR socket option. |
| STATISTICS
STATS | Write messages containing statistics when <i>tcplisten</i> terminates. The format of these statistics is undefined. STATS is a synonym. |
| USERID | Specify the user ID of the virtual machine or started task where TCP/IP runs. The default is TCPIP. |

Operation: *tcplisten* creates a socket, binds it to the specified port, writes the socket address if GETSOCKNAME is specified, and listens on the socket. *tcplisten* then performs these steps repeatedly:

1. If *tcplisten* is not a first stage, it waits for a record to arrive on the primary input stream. It terminates if the primary input stream is severed.
2. It accepts a connection. This will cause it to wait when no connection request is queued in the TCP/IP stack.
3. It performs the `givesocket()` function to allow another program to take the socket.
4. It writes an output record that describes the socket that is allocated to the connection. This record should be passed to a *tcpdata* stage without being delayed. The *tcpdata* will perform the `takesocket()` function to obtain the socket.
5. It closes the socket. If the socket has not been taken by a *tcpdata* stage, possibly because the request should be rejected, TCP/IP will now terminate the connection.
6. If *tcplisten* is not a first stage, it consumes the record on the primary input stream.

Output Record Format:

Pos	Len	Description
1	8	Check word. The constant <code>pipetcp</code> (one trailing blank).
9	8	The ID of the virtual machine or started task that runs TCP/IP. (The <code>USERID</code> operand on the <i>tcplisten</i> stage.)
17	40	The <code>clientid</code> structure filled with information about the <i>tcplisten</i> stage.
57	4	The socket number, binary.
61	4	A fullword of zeros (for the socket number in <code>takesocket</code>).
65	16	The network address of the client (the structure <code>sockaddr_in</code>).
81	4	The address of <i>tcplisten</i> 's work area.

Streams Used: Secondary streams may be defined. When the secondary output stream is connected, a record is written to it when *tcplisten* terminates after TCP/IP has reported an "ERRNO".

Record Delay: *tcplisten* does not delay the record.

Commit Level: *tcplisten* starts on commit level -10. It binds a socket to the port, verifies that its secondary input stream is not connected, and then commits to level 0.

Premature Termination: When *tcplisten* is first in the pipeline, it does not terminate normally. It terminates when it discovers that its primary output stream is not connected.

tcplisten also terminates when an error is reflected by TCP/IP (known as an ERRNO). How it terminates depends on whether the secondary output stream is defined or not.

When the secondary output stream is not connected, error messages are issued to describe the error and *tcplisten* terminates with a nonzero return code.

tcplisten

When the secondary output stream is connected, a single record is written to the secondary output stream; *tcplisten* then terminates with the return code zero. The record written contains the error number; the second word contains the symbolic name of the error number if the error number is recognised by *CMS Pipelines*. The assumption is that a REXX program will inspect the error number and decide whether it should retry the operation, discard the current transaction and retry, or give up entirely.

tcplisten also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *tcpclient*, *tcpdata*, and *udp*.

Examples: A generalised server could look like this:

```
pipe tcplisten 260 | tcpserver

/* TCPSEVER REXX */
signal on error
do forever
  'peekto'                                /* Wait for connection request */
  'addpipe *.output: | i: fanin | tcpdata | server | i:' /* subtask */
  'callpipe *: | take 1 | *:'             /* Pass one record to subtask */
  'sever output'                          /* Let server run unconnected */
end
error: exit RC*(RC<>0)
```

The four pipeline commands in the example above implement a loop that spawns a separate pipeline for each connection request. The record produced when *tcplisten* receives a connection request is passed to this pipeline, which is then cut loose to live its own independent life.

Any vetting of the client should be done before the ADDPIPE pipeline command is issued. If the client is not authorised, the input record should be consumed by a READTO pipeline command. This will cause *tcplisten* to close the socket without its being taken by the server task, and thus, the request will be rejected.

Notes:

1. *tcplisten* does not read or write data to the sockets it handles.
2. *CMS Pipelines* defines error numbers in the 5000 range in addition to the ones defined by TCP/IP:
 - 5000 (EpipeResponseTimedOut) No response was received within the interval specified by TIMEOUT.
 - 5001 (EpipeStopped) The pipeline was signalled to stop by passing a record to *pipestop* or through a similar action.
 - 5002 (EpipeSocketClosed) ONERESPONSE is specified and the socket was closed by the communications partner without it sending a response to a transaction.

CMS Pipelines also defines this error number:

 - 0000 (OKSocketClosed) The connection was closed by the communications partner. The stage is not expecting a response; that is, ONERESPONSE is omitted.
3. It is customary to terminate *tcplisten* by passing a record to a *gate* stage that is connected to the output. Be sure to connect the *gate* to both output streams when you are using the secondary output stream.

4. The file transfer protocol requires TCP/IP to assign a port number for listening; this is arcane usage.
5. By controlling the rate of arrival of input records, the pipeline programmer can avoid running the virtual machine or address space out of resources in the event of it being flooded with connection requests.
6. A validation stage may be inserted immediately after the *tcplisten* stage. Its purpose is to pass the input record to the output when the request is authorized, but to consume the record when the request should be rejected.

Return Codes: When the secondary output stream is defined and *tcplisten* terminates due to an error that is reported by TCP/IP as an ERRNO, *tcplisten* sets return code 0 because the error information is available in the record that is written to the secondary output stream. When *tcplisten* terminates because of some other error (for example, if it could not connect to the TCP/IP address space), the secondary output stream is ignored and the return code is not zero, reflecting the number of the message issued to describe this error condition.

threeway—Split record three ways

threeway splits the record in three parts: The part up to the specified input range is written to the primary output stream; the contents of the input range are then written to the secondary output stream; finally, the remainder of the record is written to the tertiary output stream.

▶▶—THREEWAY—*inputRange*—▶▶

Type: Gateway.

Syntax Description: Specify an input range.

Operation: The input record is split before and after the specified input range. The part up to the beginning of the range is written to the primary output stream; the contents of the input range is then written to the secondary output stream; and the balance of the record is then written to the tertiary output stream. Finally, the input record is consumed.

Streams Used: Three streams must be defined. Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *threeway* strictly does not delay the record.

Commit Level: *threeway* starts on commit level -2. It verifies that the primary input stream is the only connected input stream and then commits to level 0.

Premature Termination: *threeway* terminates when it discovers that no output stream is connected.

See Also: *chop* and *substring*.

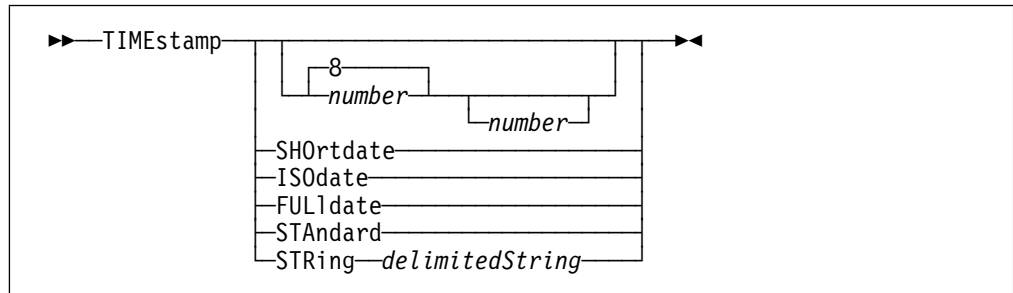
Notes:

1. *3way* is a synonym for *threeway*.
2. A null record is written when the corresponding part of the record is not present.

timestamp

timestamp—Prefix the Date and Time to Records

timestamp prefixes each record with a timestamp showing when the record was processed by *timestamp*.



Type: Filter.

Syntax Description: The arguments are optional. The formatted timestamp can be the “raw” 16-byte sorted timestamp, a predefined format, or a custom format.

number The first number specifies the position, relative to the end of the formatted timestamp, of the first character to include; it is quietly limited to 16. The default is 8, which omits the date when the second number is omitted. The second number specifies the count of characters to include. The default is the same as the first number; it is quietly restricted to 16 minus the first number.

FULLDATE The file’s timestamp is formatted in the American format, with the century: 3/09/1946 23:59:59.

ISODATE The file’s timestamp is formatted with the century in one of the formats approved by the International Standardisation Organisation: 1946-03-09 23:59:59.

SHORTDATE The file’s timestamp is formatted in the American format, without the century: 3/09/46 23:59:59.

STANDARD The file’s timestamp is formatted as a single word in a form that can be used for comparisons: 19460309235959.

STRING Specify custom timestamp formatting, similar to the POSIX `strftime()` function. The delimited string specifies formatting as literal text and substitutions are indicated by a percentage symbol (%) followed by a character that defines the substitution. These substitution strings are recognised by *timestamp*:

:	%	A single %.
:	%Y	Four digits year including century (0000-9999).
:	%y	Two-digit year of century (00-99).
:	%m	Two-digit month (01-12).
:	%n	Two-digit month with initial zero changed to blank (1-12).
:	%d	Two-digit day of month (01-31).
:	%e	Two-digit day of month with initial zero changed to blank (1-31).
:	%j	Julian day of year (001-366).
:	%H	Hour, 24-hour clock (00-23).
:	%k	Hour, 24-hour clock first leading zero blank (0-23).
:	%M	Minute (00-59).
:	%S	Second (00-60).
:	%F	Equivalent to %Y-%m-%d (the ISO 8601 date format).
:	%T	Short for %H:%M:%S.
:	%t	Tens and hundredth of a second (00-99).

The length of the formatted timestamps and the equivalent string are:

Standard	14	%Y%m%d%H%M%S
ISO	19	%F %T
Full	19	%n/%d/%Y %k:%M:%S
Short	17	%n/%d/%y %k:%M:%S

Operation: A 16-character timestamp is developed whenever a record has been read. It consists of the year (including the century), the month, day, hour, minute, second, and hundredth of a second. It is formatted as specified by the operands.

Record Delay: *timestamp* strictly does not delay the record.

Premature Termination: *timestamp* terminates when it discovers that its output stream is not connected.

See Also: *spec*.

Examples: To see what the current time or the current date and time is:

```

pipe literal | timestamp | console
▶14503501
▶Ready;
pipe literal | timestamp 16 | console
▶2020042914503502
▶Ready;
pipe literal | timestamp string /Day %e of Month %n in %Y at %H/ | ...
... console
▶Day 29 of Month 4 in 2020 at 14
▶Ready;

```

To timestamp records that are logged in a service machine:

```
'pipe starmsg | ... | timestamp 16 | >> service log a'
```

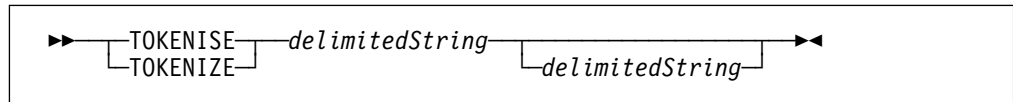
Notes:

1. *timestamp* obtains the local time. Use *spec* TOD C2T 1 to obtain the current time in UTC, assuming, of course, that the TOD clock is set to the standard epoch.

tokenise

tokenise—Tokenise Records

tokenise splits an input record into tokens, writing an output record for each. Blanks always delimit tokens; they are discarded. The specified string contains additional characters; when an input record contains one of these characters, the character is written by itself in an output record.



Type: Filter.

Syntax Description: One delimited string is mandatory; one is optional.

Operation: A line is written for each token in the input record. Blanks always delimit tokens. The first delimited string lists characters that delimit other tokens.

The second argument string, if present, is written as a separate record after each input line is processed.

Record Delay: *tokenise* does not delay the last record written for an input record.

Premature Termination: *tokenise* terminates when it discovers that its output stream is not connected.

Examples: To tokenise according to the CMS rules, adding a blank line after the tokens of an input record:

```
...| tokenise /()/ / / | pad 8 | chop 8 | xlate upper |...
```

To tokenise without padding, chopping, or translation:

```
pipe literal apples = bananas(cherries+dates) | tokenise /()=+/ | ...  
... console
```

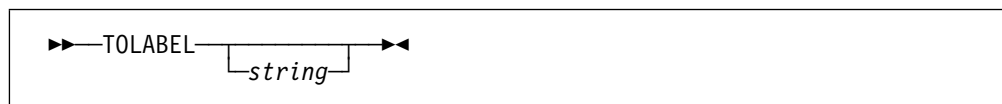
```
▶apples  
▶=  
▶bananas  
▶(  
▶cherries  
▶+  
▶dates  
▶)  
▶Ready;
```

Notes:

1. Tokens are neither padded, truncated, nor translated to upper case.

tolabel—Select Records to the First One with Leading String

tolabel selects input records up to the first one that begins with the specified string. That record and the records that follow are discarded.



Type: Selection stage.

Syntax Description: A string is optional. The string starts after exactly one blank character. Leading and trailing blanks are significant.

Operation: Characters at the beginning of each input record are compared with the argument string. Any record matches a null argument string. A record that is shorter than the argument string does not match.

tolabel copies records up to (but not including) the matching one to the primary output stream, or discards them if the primary output stream is not connected. If the secondary output stream is defined, *tolabel* then passes the remaining input records to the secondary output stream.

The matching record stays in the pipeline if the secondary output stream is not defined; it can be read again if the current pipeline is defined with CALLPIPE.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. If the secondary output stream is defined, *tolabel* severs the primary output stream before it passes the remaining input records to the secondary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *tolabel* strictly does not delay the record.

Commit Level: *tolabel* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *tolabel* terminates when it discovers that no output stream is connected.

Converse Operation: *frlabel*.

See Also: *between*, *inside*, *notin*, *outside*, *strtolabel*, and *whilelabel*.

Examples: To load records up to the first one beginning with '.end':

```
/* Load batch of records into stem */
'callpipe *: | tolabel .end| stem todo.'
```

tolabel is before the *stem* stage that loads the variables; all lines would be processed if the order of the stages were reversed.

Notes:

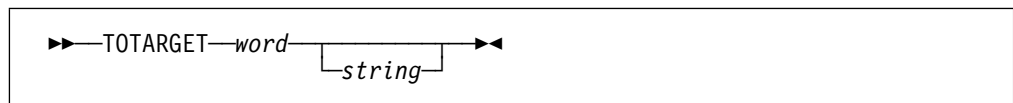
1. Use *strtolabel* with ANYCASE for a caseless compare.
2. Remember that REXX continuation functionally replaces a trailing comma with a blank.

totarget

Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

totarget—Select Records to the First One Selected by Argument Stage

The argument to *totarget* is a stage to run. *totarget* passes records to this stage until the stage produces an output record on its primary output stream. The trigger record and the remaining input are then shorted to the secondary output stream (if it is defined). Records that are rejected by the argument stage are passed to the primary output stream.



Type: Control.

Syntax Description: The argument string is the specification of a selection stage. The stage must support a connected secondary output stream. If the secondary input stream to *totarget* is connected, the argument stage must also support a connected secondary input stream.

Streams Used: Two streams may be defined.

Record Delay: *totarget* does not add delay.

Commit Level: *totarget* starts on commit level -2. It issues a subroutine pipeline that contains the argument stage. This subroutine must commit to level 0 in due course.

Premature Termination: *totarget* terminates when it discovers that no output stream is connected.

Converse Operation: *frtarget*.

See Also: *gate* and *predselect*.

Examples: To pass to the primary output stream all records up to the first one that contains a string and to pass the remaining records to the secondary output stream:

```
/* Totarget example */
'callpipe (end ? name TOTARGET)',
  |*:', /* Connect to input */
  |f: totarget locate /abc/', /* Look for it */
  |*.output.0:', /* Up to target */
  |?f:',
  |*.output.1:' /* Target and rest */
exit RC
```

Notes:

1. It is assumed that the argument stage behaves like a selection stage: the stage should produce without delay exactly one output record for each input record; it should terminate without consuming the current record when it discovers that its output streams are no longer connected. However, for each input record the stage can produce as many records as it pleases on its secondary output stream; it can delete records. The stage

should not write a record first to its secondary output stream and then to its primary output stream; this would cause the trigger record to be written to both output streams.

If the argument stage has delayed record(s) (presumably by storing them in an internal buffer) at the time it writes a record to its primary output stream, it will not be able to write these records to any output stream; the streams that are connected to the two output streams are severed when the argument stage writes a record to its primary output stream. End-of-file is reflected on this write. The records held internally in the argument stage will of necessity be lost when the stage terminates.

2. The argument string to *totarget* is passed through the pipeline specification parser only once (when the scanner processes the *totarget* stage), unlike the argument strings for *append* and *preface*.
3. *totarget* is implemented using *fillup* and *fanoutwo*. The stage under test has only primary streams defined. The primary output stream is connected to a stage that reads a record without consuming it and then terminates. This means that any usage that depends on the secondary stream in the stage under test, will fail.

Return Codes: If *totarget* finds no errors, the return code is the one received from the selection stage.

trackblock—Build Track Record

trackblock builds a standard format track from its constituent home address and data records (blocks).

►►—TRACKBLOCK—◄◄

Type: Arcane filter.

Input Record Format: For each track, the home address (FCCHH) is followed by as many records as there are data records (blocks) on the track. The data records begin with an 8-byte count area (CCHHRKDD). The end of track record is optional.

Output Record Format: An 8-byte check word that contains the string `fp1track`. Then follows the contents of the track as read by the read track CCW. This contains a 5-byte home address and a number of records starting with record 0 (if the track is formatted in the standard format). Each record contains one to three parts:

- The count area. This is an 8-byte area that contains the cylinder, head, and record number followed by the size of the key area and the data area (CCHHRKDD).
- Key area, if present.
- Data area, if present.

Record Delay: When a track contains an end of track record, the output record is not delayed relative to the end of track record; otherwise the output is delayed until the arrival of the next home address record.

Premature Termination: *trackblock* terminates when it discovers that its output stream is not connected.

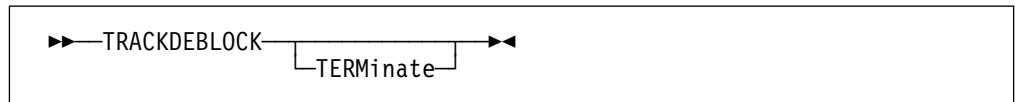
Converse Operation: *trackdeblock*.

Notes:

1. For ECKD devices with more than 65519 cylinders, Extended Address Volumes format specifies how a 28-bit cylinder number and 4-bit track number are encoded in the 32-bit word referred to as CCHH.

trackdeblock—Deblock Track

trackdeblock splits the track into its parts, the home address and data records.



Type: Arcane filter.

Syntax Description: A keyword is optional.

TERMINATE Append an end of track marker of 8 bytes of binary ones.

Input Record Format: An 8-byte check word that contains the string fpltrack. Then follows the contents of the track as read by the read track CCW. This contains a 5-byte home address and a number of records starting with record 0 (if the track is formatted in the standard format). Each record contains one to three parts:

- The count area. This is an 8-byte area that contains the cylinder, head, and record number followed by the size of the key area and the data area (CCHHRKDD).
- Key area, if present.
- Data area, if present.

Output Record Format: For each track, the home address (FCCHH) is followed by as many records as there are data records (blocks) on the track. The data records begin with an 8-byte count area (CCHHRKDD). When TERMINATE is specified, trackdeblock writes a record containing 8 bytes of all one bits as an end of track marker.

Record Delay: trackdeblock does not delay the record.

Premature Termination: trackdeblock terminates when it discovers that its output stream is not connected.

Converse Operation: trackblock.

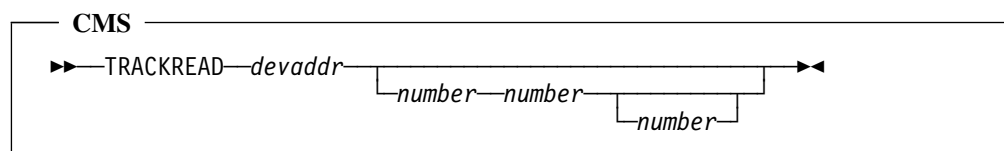
See Also: tracksquish and ckddblock.

Notes:

1. The beginning of a new track can be inferred by the length of the 5-byte home address; all other records contain at least the eight bytes of their count area.
2. For ECKD devices with more than 65519 cylinders, Extended Address Volumes format specifies how a 28-bit cylinder number and 4-bit track number are encoded in the 32-bit word referred to as CCHH.

trackread—Read Full Tracks from ECKD Device

trackread reads the contents of one or more tracks from a disk that supports the Extended Count Key Data (ECKD) command set, such as an IBM 3390.



Type: Arcane device driver.

Syntax Description: Specify the device number and an initial range of tracks to read.

<i>devaddr</i>	The virtual device number of the disk to read.
<i>number</i>	The starting cylinder number.
<i>number</i>	The starting track number.
<i>number</i>	The number of tracks to read. The default is 1. Specify an asterisk (*) to read to the end of the device.

Operation: *trackread* verifies the device number as part of the syntax *trackread* writes one output record for each track that was read from the disk.

Input Record Format: Additional extents to be read. Specify two or three blank-delimited words: the initial cylinder and track, and the count of tracks or an asterisk.

Output Record Format: An 8-byte check word that contains the string `fp1track`. Then follows the contents of the track as read by the read track CCW. This contains a 5-byte home address and a number of records starting with record 0 (if the track is formatted in the standard format). Each record contains one to three parts:

- The count area. This is an 8-byte area that contains the cylinder, head, and record number followed by the size of the key area and the data area (CCHHRKDD).
- Key area, if present.
- Data area, if present.

Record Delay: *trackread* does not delay the record.

Premature Termination: *trackread* terminates when it discovers that its primary output stream is not connected.

Converse Operation: *trackwrite*.

See Also: *tracksquish* and *trackverify*.

Examples: To save a disk image:

```
pipe trackread 190 0 0 * | tracksquish | pack | > 190 image a
```

Notes:

1. The disk needs not be accessed or even supported by CMS.
2. For ECKD devices with more than 65519 cylinders, Extended Address Volumes

! format specifies how a 28-bit cylinder number and 4-bit track number are encoded in
! the 32-bit word referred to as CCHH.

tracksquish—Squish Tracks

tracksquish reduces the size of a standard track that is formatted, but unused.



Type: Arcane filter.

Operation: tracksquish passes records that already are in the squished format; it reduces the size of a track in the standard format (as written by trackread) that is formatted by CP or CMS.

Input Record Format: An 8-byte check word that contains the string fpltrack. Then follows the contents of the track as read by the read track CCW. This contains a 5-byte home address and a number of records starting with record 0 (if the track is formatted in the standard format). Each record contains one to three parts:

- The count area. This is an 8-byte area that contains the cylinder, head, and record number followed by the size of the key area and the data area (CCHHRKDD).
- Key area, if present.
- Data area, if present.

Output Record Format: An 8-byte check word that contains the string fplsquis. The remainder of the record is unspecified.

Record Delay: tracksquish does not delay the record.

Premature Termination: tracksquish terminates when it discovers that its primary output stream is not connected.

Converse Operation: trackxpan.

See Also: trackdeblock.

Notes:

1. tracksquish does not compress the track.
2. For ECKD devices with more than 65519 cylinders, Extended Address Volumes format specifies how a 28-bit cylinder number and 4-bit track number are encoded in the 32-bit word referred to as CCHH.

trackverify—Verify Track Format

trackverify verifies that the input records contain valid tracks in the standard format (as produced by trackread). It issues diagnostic messages and terminates as soon as it finds a track that is not valid.



Type: Arcane filter.

Input Record Format: An 8-byte check word that contains the string `fp|track`. Then follows the contents of the track as read by the read track CCW. This contains a 5-byte home address and a number of records starting with record 0 (if the track is formatted in the standard format). Each record contains one to three parts:

- The count area. This is an 8-byte area that contains the cylinder, head, and record number followed by the size of the key area and the data area (CCHHRKDD).
- Key area, if present.
- Data area, if present.

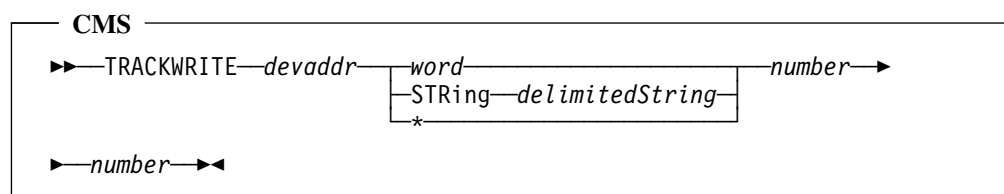
Streams Used: Records are read from the primary input stream; no other input stream may be connected. *trackverify* does not produce output.

Notes:

1. For ECKD devices with more than 65519 cylinders, Extended Address Volumes format specifies how a 28-bit cylinder number and 4-bit track number are encoded in the 32-bit word referred to as CCHH.

trackwrite—Write Full Tracks to ECKD Device

trackwrite writes the contents of one or more tracks to a disk that supports the Extended Count Key Data (ECKD) command set, such as an IBM 3390.



Type: Arcane device driver.

Placement: *trackwrite* must not be a first stage.

Syntax Description: Specify the device number, the current label on the device, and the first and last cylinder in the writable extent.

devaddr The virtual device number of the disk to write.

word The current volume label on the device.

STRING

*

- A *word*, which is made upper case.
- The keyword STRING followed by a *delimitedString* for a label that contains characters in lower case or blanks.
- An asterisk to indicate that no label is present, for example, on a fresh temporary disk.

number The first writable cylinder number.

number The last writable cylinder number.

The first and last cylinders specify the extent into which tracks are written; the actual track address is obtained from the input record.

Operation: trackwrite verifies the device number and label as part of the syntax check.

Input Record Format: trackwrite supports input records in the format produced by both the standard track format (trackread) and the squished track format (tracksquish).

The standard track format contains an 8-byte check word that contains the string fpltrack. Then follows the contents of the track as read by the read track CCW. This contains a 5-byte home address and a number of records starting with record 0 (if the track is formatted in the standard format). Each record contains one to three parts:

- The count area. This is an 8-byte area that contains the cylinder, head, and record number followed by the size of the key area and the data area (CCHHRKDD).
- Key area, if present.
- Data area, if present.

The squished track format contains an 8-byte check word that contains the string fplsquis. The remainder of the record is unspecified.

Streams Used: trackwrite passes the input to the output.

Record Delay: trackwrite strictly does not delay the record.

Converse Operation: trackread.

See Also: trackxpan.

Notes:

1. For ECKD devices with more than 65519 cylinders, Extended Address Volumes format specifies how a 28-bit cylinder number and 4-bit track number are encoded in the 32-bit word referred to as CCHH.

trackxpan—Unsquish Tracks

trackxpan expands a squished track to the standard format.



Type: Arcane filter.

Operation: trackxpan passes records that already are in the standard format; it expands squished records to the standard format.

Input Record Format: An 8-byte check word that contains the string fplsquis. The remainder of the record is unspecified.

Output Record Format: An 8-byte check word that contains the string fpltrack. Then follows the contents of the track as read by the read track CCW. This contains a 5-byte home address and a number of records starting with record 0 (if the track is formatted in the standard format). Each record contains one to three parts:

- The count area. This is an 8-byte area that contains the cylinder, head, and record number followed by the size of the key area and the data area (CCHHRKDD).
- Key area, if present.

first 36 bytes contain a header. Refer to the CP Programming Services manual for further information.

When CP is specified, it is verified that the SPOOL file contains CP trace data. The output record is 32 or 60 bytes long and contains the full 8-byte TOD timestamp of the entry. When LOCALTIME is specified, the timestamp is adjusted with the local time zone, which is present in the input record. The leftmost bit of the CPUID field indicates a double length entry created from a 64-byte trace table entry. The last fullword of the 64-byte trace table entry is not present.

When TRSOURCE is specified, it is verified that the file contains TRSOURCE data. The individual data records are deblocked.

Commit Level: *trfread* starts on commit level -2. It opens the SPOOL file and then commits to level 0.

Premature Termination: *trfread* terminates when it discovers that no output stream is connected.

See Also: *reader*.

tso—Issue TSO Commands, Write Response to Pipeline

tso issues TSO commands and captures the command response, which is then written to the output rather than being displayed on the terminal.



Type: Host command interface.

Syntax Description:

Operation: The argument string (if present) and input lines are issued to TSO. The response from the TSO commands is not written to the terminal. The response from each command is buffered until the command ends; the response is then written to the output.

Record Delay: *tso* writes all output for an input record before consuming the input record.

Premature Termination: *tso* terminates as soon as a negative return code is received.

See Also: *command*.

Examples: To display data set allocation in a more readable format:

```

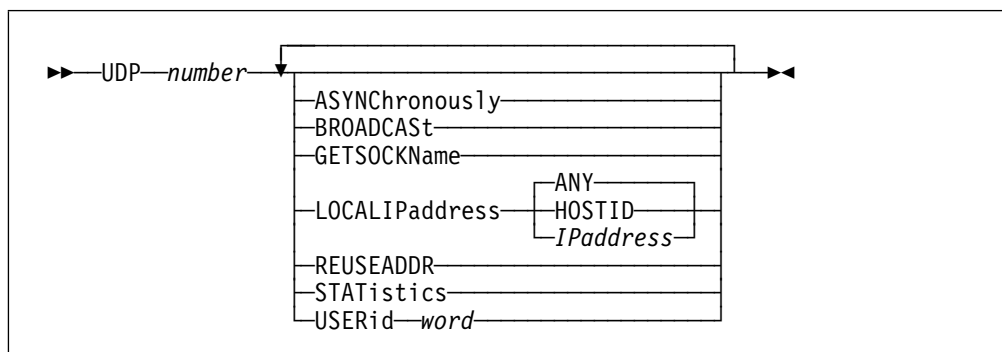
pipe tso lista st | drop 1 | spec 1-* 15 read 1.10 1 | take 3 | console
▶ STEPLIB      DPJOHN.TSO.LOAD
▶ SYSHELP      SYS1.HELP
▶ PIPEHELP     DPJOHN.PIPE.HELPLIB
▶ READY
  
```

Notes:

1. *tso* is implemented as a REXX program that uses the OUTTRAP function. Thus, *tso* can trap only what can be trapped by OUTTRAP.

udp—Read and Write an UDP Port

When *udp* is first in a pipeline it reads requests from the specified port into the pipeline. When *udp* is not first in a pipeline it sends its input records to a port and writes the responses into the pipeline.



Type: Device driver.

Syntax Description: A decimal number is required; it can be from 0 to 65535 (inclusive). The number specifies the number of the receive port. Zero indicates that you wish TCP/IP to assign a port number.

ASYNCHRON	This keyword is optional when <i>udp</i> is not first in a pipeline; it is not allowed when <i>udp</i> is first in a pipeline. <i>udp</i> should receive independently of how it sends. By default, <i>udp</i> sends one record and then waits for a response.
BROADCAST	Turn on the BROADCAST socket option.
GETSOCKNAME	Write the contents of the socket address structure to the primary output stream after the socket is bound.
LOCALIPADDR	Specify the local IP address to be used when binding the socket. The default, ANY, specifies that TCP/IP may use any interface address. (An IP address of binary zeros is used to bind the socket.) HOSTID specifies that TCP/IP should use the IP address that corresponds to the host name. Specify the dotted-decimal notation or (on CMS) the host name for a particular interface to be used.
REUSEADDR	Turn on the REUSEADDR socket option.
STATISTICS STATS	Write messages containing statistics when <i>udp</i> terminates. The format of these statistics is undefined. STATS is a synonym.
USERID	Specify the user ID of the virtual machine or started task where TCP/IP runs. The default is TCPIP.

Operation: When *udp* is first in a pipeline, it waits for messages on the port and writes each message as an output record as it arrives.

When *udp* is not first in a pipeline and the keyword ASYNCHRONOUSLY is specified, it

udp

reads input records and sends them to the destination specified in the record. It writes arriving messages to the output as they arrive.

When ASYNCHRONOUSLY is not specified, *udp* loops performing these steps:

1. Ensure that the output stream is still connected and terminate if not.
2. Read a record (if *udp* is not first in the pipeline). If the record is 24 bytes or longer, the datagram is sent to the port specified in the record. If the first four bytes are nonzero they specify a timeout value in seconds.
3. Wait for a datagram to arrive at the port specified in the arguments or for a timeout to occur. When a datagram is received, it is written to the pipeline. If *udp* is not first in the pipeline, a null record is written in case of a timeout.

Input Record Format: The input record must be four bytes long (to indicate that UDP should listen only for the indicated period of time) or at least 24 bytes long (to specify a timeout value and a port to receive the datagram). The input record contains information required by the SENDTO IUCV socket function:

Offs	Len	Contents
0	4	Timeout value in seconds (binary). A value of zero specifies that <i>udp</i> should not wait for a reply. This value is ignored when ASYNCHRONOUSLY is specified.
4	4	Flag bytes. Usually binary zeros. The value 4 specifies MSG_DONTROUTE, which is used by diagnostic or routing programs.
8	16	Network address of the destination port. This consists of the addressing family (X'0002' to indicate AF_INET), the port number (16 bits), the Internet adapter address (32 bits), and eight bytes of zeros.
24	n	The datagram to be sent. For TFTP, it begins with a two-byte operation code. The Internet does not limit datagrams to a specific size, but suggests that networks and gateways should be prepared to handle datagrams of up to 576 octets without fragmenting them. (From Douglas E Comer, <i>Internetworking with TCP/IP</i> , Prentice-Hall, 1988.)

Output Record Format: A null record indicates a timeout; no datagram was received. A record that is not null contains a datagram received:

Offs	Len	Contents
0	16	Network address of the origin port. This consists of the address family (X'0002'), the port number (16 bits), the Internet adapter address (32 bits), and eight bytes of zeros.
16	n	The datagram received. For TFTP, it begins with a two-byte operation code.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. When the secondary output stream is connected, a record is written to it when *udp* terminates after TCP/IP has reported an “ERRNO”.

Commit Level: *udp* starts on commit level -10. It creates a socket, verifies that its secondary input stream is not connected, and then commits to level 0.

Premature Termination: *udp* terminates when it discovers that its primary output stream is not connected.

udp also terminates when an error is reflected by TCP/IP (known as an ERRNO). How it terminates depends on whether the secondary output stream is defined or not.

When the secondary output stream is not connected, error messages are issued to describe the error and *udp* terminates with a nonzero return code.

When the secondary output stream is connected, a single record is written to the secondary output stream; *udp* then terminates with the return code zero. The record written contains the error number; the second word contains the symbolic name of the error number if the error number is recognised by *CMS Pipelines*. The assumption is that a REXX program will inspect the error number and decide whether it should retry the operation, discard the current transaction and retry, or give up entirely.

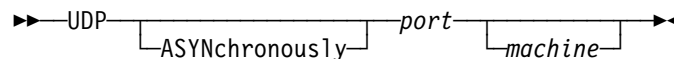
udp also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

Examples: To write TFTP requests to the pipeline:

```
pipe udp 69 | ....
```

Notes:

1. The User Datagram Protocol is said to be connectionless. That is, it is like one virtual machine sending a message to other virtual machines (as opposed to having an IUCV connection); any response is generated by the receiver of its own accord.
2. While TCP/IP tries to deliver messages as best it can, the User Datagram Protocol does not specify that messages must arrive in the order they are sent; nor does it provide for notification of lost messages. A protocol must be defined at a higher level to implement error recovery. (This is often called “the end to end argument”.)
3. For compatibility with earlier releases, *udp* also accepts an abbreviated format for its arguments:



4. A null packet from the net contains 16 bytes of socket address of the origin, whereas a timeout causes a null record to be written.

Return Codes: When the secondary output stream is defined and *udp* terminates due to an error that is reported by TCP/IP as an ERRNO, *udp* sets return code 0 because the error information is available in the record that is written to the secondary output stream. When *udp* terminates because of some other error (for example, if it could not connect to the TCP/IP address space), the secondary output stream is ignored and the return code is not zero, reflecting the number of the message issued to describe this error condition.

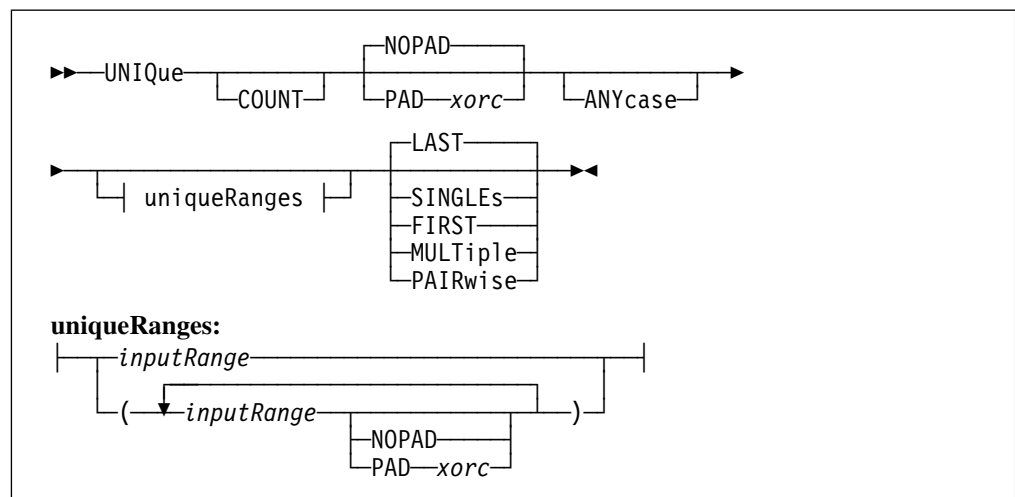
unique

unique—Discard or Retain Duplicate Lines

unique compares key fields in pairs of input records and selects records based on the result of this comparison and specified options. It can select:

- The first record in a run of records having a particular key, discarding further records with this key.
- The last record in a run of records having a particular key, discarding the leading records with this key.
- Records for which there are no duplicates.
- Records for which there are duplicates.
- Pairs of records that are not duplicates.

The sequence number of the record within a run of records with identical keys can be prefixed to the record; when the last occurrence of a record is selected, this sequence number becomes the count of records with that particular key.



Type: Selection stage.

Syntax Description: The keyword `NOPAD` specifies that key fields that are partially present must have the same length to be considered equal; this is the default. The keyword `PAD` specifies a pad character that is used to extend the shorter of two key fields.

The keyword `ANYCASE` specifies that case is to be ignored when comparing fields; the default is to respect case. An optional input range or a list of input ranges in parentheses may be followed by up to two keywords. Each input range may be followed by the keywords `PAD` or `NOPAD` to specify padding for this particular field. For compatibility with the past, *unique* also accepts two optional keywords followed by an optional input range or a list of input ranges in parentheses. `PAIRWISE` cannot be specified with `COUNT`.

Operation: Records are written to the primary output stream or the secondary output stream, depending on the contents of the column ranges and the option specified.

1. When `PAIRWISE` is omitted:

For each record on the primary input stream, the contents of the column ranges specified (the complete record by default) are compared with the contents of the corresponding ranges in the following record. A run of records having the same contents of the column ranges comprises a set of duplicate records. When the keyword `NOPAD` is specified (or the padding option is omitted), a position not present in a record has a

value that is not equal to any possible contents of a position that is present. When PAD is specified, short key fields are extended with the pad character for purposes of comparison. Otherwise, when a range is partially present, two records must be the same length and contain the same data within the range to compare equal.

When the keyword COUNT is used, each record is prefixed with a 10-character field indicating its position in a run of equal records (starting with 1). When combined with the default option LAST, COUNT sets the count of identical records in the first 10 positions of the record written to the primary output stream.

For a set of duplicate records, keywords determine which records are selected:

SINGLES	Runs that consist only of one record are copied to the primary output stream. Thus, only truly unique records are selected.
FIRST	The first record of a run is copied to the primary output stream. That is, all singles and the first record of each set of duplicates are selected.
LAST	The last record of a run is copied to the primary output stream. Thus, singles and the last record of a set of duplicates are selected. (This is the default.)
MULTIPLE	Runs that contain more than one record are written to the primary output stream. That is, only truly duplicate records are selected.

- When PAIRWISE is specified, a record is read and compared with the following record. Both records are written to the primary output stream when their key fields are not equal.

Records that are not written to the primary output stream are written to the secondary output stream (or discarded if it is not connected). For example, *unique* SINGLES writes complete sets of duplicates on the secondary output stream.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. Output is written to the primary output stream and the secondary output stream depending on options.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. Except for FIRST, which does not delay the record, and PAIRWISE, which delays the first record of a pair but not the second, *unique* delays one record.

Commit Level: *unique* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *unique* terminates when it discovers that no output stream is connected.

See Also: *sort*.

Examples: To list files that are on both of two minidisks or accessed directories:

unpack

```
/* BOTHDISK REXX -- Select files on two disks                                */
signal on novalue
arg model fm fn ft .
If fm=''
Then signal tell
parse value fn ft '* *' with fn ft .
'callpipe',
  '| literal LISTFILE' fn ft model '(NOH',
  '| command LISTFILE' fn ft fm '(NOH',
  '| sort',                                /* Order                                */
  '| unique 1.17 mult',                    /* Get duplicate files                  */
  '| unique 1.17 first',                   /* Just first occurrence                */
  '| *:'
exit RC

tell:
say 'Usage: BOTHDISK <fm1> <fm2> [<fn> [<ft>]]'
```

Notes:

1. *unique* compares only adjacent records. It is normal to sort the file earlier in the pipeline.
2. Use *sort* UNIQUE instead of a cascade of *sort* and *unique* when the file has many duplicate records and you do not wish to process the duplicates further.
3. Unless ANYCASE is specified, key fields are compared as character data using the IBM System/360 collating sequence.
4. Use *spec* (or a REXX program) for example to put a sort key in front of the record if you wish, for instance, to use a numeric field that is not aligned to the right within a column range. Such a temporary sort key can be removed with *substr* for example after the records are written by *unique*.
5. Use *xlate* to change the collating sequence of the file.
6. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.
7. *unique* has no option to specify the inverse of PAIRWISE. Use *not unique* to swap the contents of the output streams.

unpack—Unpack a Packed File

unpack turns a packed file back into plain records; it does not modify a file that is not packed.

▶▶—UNPACK—◀◀

Type: Filter.

Operation: A file not in the packed format created by XEDIT, COPYFILE, or *pack* is passed through unmodified. A null record is interpreted as end-of-file and the next record is inspected to see if that is the beginning of a packed file.

Streams Used: Records are read from the primary input stream and written to the primary output stream.

Record Delay: *unpack* delays input records as required to build an output record. The delay is unspecified.

Premature Termination: *unpack* terminates when it discovers that its output stream is not connected.

Converse Operation: *pack*.

Examples: To unpack a file:

```

pipe cms listfile pipodent copy * ( format | console
▶FILENAME FILETYPE FM FORMAT LRECL
▶PIPODENT COPY      K1 F      1024
▶Ready;
pipe < pipodent copy | unpack | chop 72 | console
▶*COPY PGMID
▶ GBLC      &PGMID,&MODULE
▶&PGMID SETC 'PIP'
▶Ready;

```

Notes:

1. *unpack* can unpack files that XEDIT and COPYFILE cannot cope with.
2. *unpack* is “safe” after <. It unpacks a file if it is packed, and passes a file that is not packed through unchanged; it does not issue a diagnostic if the file is not packed.

untab—Replace Tabulate Characters with Blanks

untab expands tab characters in the record to blanks to line up columns.



Type: Filter.

Syntax Description: No arguments are required. A single negative number or a list of positive numbers may be specified.

A list of positive numbers enumerates the tab stops; the numbers may be in any order. The smallest number specifies where the left margin is; use 1 to put the left margin at the beginning of the record.

A negative number specifies a tab stop in column 1, and for each *n* columns.

The default is -3, which is equivalent to 1 4 7 ...

Record Delay: *untab* strictly does not delay the record.

Premature Termination: *untab* terminates when it discovers that its output stream is not connected.

Converse Operation: *retab*.

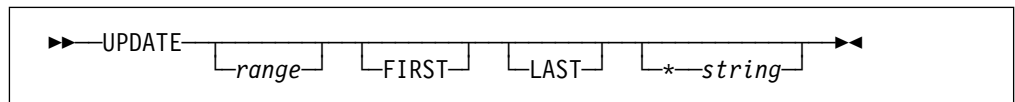
update

Examples: To expand a field read from a 3270:

```
/* Expand input */  
'callpipe var data | untab | var data'
```

update—Apply an Update File

update applies an update in the format defined for the CMS command UPDATE. A cascade of *update* stages can be used to apply several updates to a file.



Type: Gateway.

Syntax Description: Arguments are optional. A column range may be followed by two keywords. An asterisk indicates the beginning of a comment string.

When present as the first argument, a column range specifies the location of the sequence field; the default is 73-80. The maximum length of the sequence field is 15.

FIRST specifies that the filter is the first (or only) in a cascade of updates; it is ensured that the input file is correctly sequenced.

LAST specifies that *update* output sequencing is not to be verified. This lets you generate an updated file without sequence numbers. Note that this option should be used only when it is desired to suppress sequence numbering for the lines added by the last update applied.

Comments may be entered at the end of the argument string following an asterisk. This field can be used for the name of the update file being applied to identify the specific stage in error messages issued by *CMS Pipelines*. The comment string is not used by *update*.

Operation: The master file is read from the primary input stream and updated with the update file from the secondary input stream. The updated file is written to the primary output stream and the update log is written to the secondary output stream.

Control cards supported are the ones defined for UPDATE when using full sequence numbers (SEQ8).

Messages for errors that do not terminate *update* during processing are logged to the update log rather than being written to the terminal.

Streams Used: Two streams must be defined. Records are read and written on the primary stream and the secondary stream.

Record Delay: It is unspecified if *update* delays records. Applications should be written to tolerate if *update* does not delay records.

Premature Termination: *update* terminates when it discovers that either of its output streams is not connected. Connect the secondary output stream to *hole* to discard the update log.

Examples: To apply two updates to a file and discard the update logs:

```

/* Update file with two updates */
'PIPE (end ? name UPDATE)',
'| < source file',
'| u1: update first',
'| u2: update last',
'| > $source file a fixed',
'? < first update',
'| u1: ' ,
'| hole',
'? < second update',
'| u2: ' ,
'| hole'

```

The global options define the question mark as the end character. The first pipeline reads the source file and passes it through the two *update* stages. The second pipeline reads the first update file and passes it to the first *update* stage (because the label *u1:* refers back to the first *update*); the update log is discarded in *hole*. Likewise, the third pipeline reads the second update file into the second *update* stage.

Notes:

1. *update* is intended to apply an update created with XEDIT using the CTL option. Updates are applied in parallel when *update* stages are cascaded. *update* may treat some errors differently than the CMS command UPDATE does.
2. Use a cascade of *update* stages to apply several updates to a source file.

Return Codes: The following return codes are reflected when errors have been noted in the update log; when multiple errors are detected, the final return code is the highest one encountered.

- 4 Sequence request that is not first in an update file.
- 4 Sequence error in the input master file.
- 8 Trouble with the sequence control card: Start or increment is not numeric.
- 8 Sequence error in the output master file.
- 12 Trouble with the sequence number field. A sequence number is not numeric, is missing, is longer than the sequence field width; or the dollar sign is missing.
- 12 No record is found with the required sequence number.
- 32 Unsupported or missing control card.

urldeblock—Process Universal Resource Locator

urldeblock processes escape sequences and line end sequences in Universal Resource Locators. The input and output records are ASCII unless EBCDIC is specified.



Type: Filter.

Operation: Most characters of a URL are written to the output record unchanged. These characters are processed specially:

Char	Dec	Description
&	38	(Ampersand or semicolon.) Split the record. The character is discarded.
;	59	
+	43	(Plus.) Substitute a blank.
%	37	(Percent.) Hexadecimal escape sequence. The next two ASCII characters contain the hexadecimal value to be used. For example, %2b (in ASCII, that is, X'253262') becomes an ASCII plus.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *urldeblock* does not delay the record.

Premature Termination: *urldeblock* terminates when it discovers that its output stream is not connected.

Examples:

```

pipe < sample url | join | xlate e2a | urldeblock | xlate a2e | console
▶name=Craig R. Doe
▶class=90
▶grad=
▶email=xyz@prodigy.com
▶bdate=03/15/68
▶url=
▶Ready;
    
```

The file SAMPLE URL contains:

```

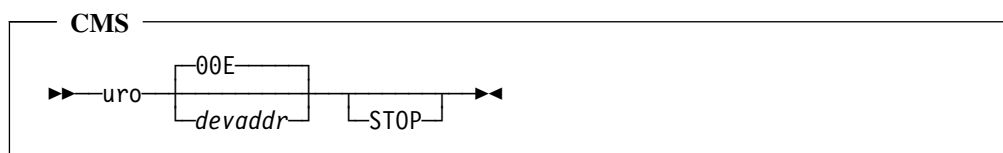
name=Craig%20R.%20Doe&class=90&grad=&email=xyz@prodigy.com&
bdate=03%2F15%2F68&url=
    
```

Notes:

1. Unlike most *CMS Pipelines* built-in programs, *urldeblock* performs its operation in the ASCII domain. If the input record has already been translated to EBCDIC by a gateway, it must be translated back to ASCII before it is passed to *urldeblock*.
2. The EBCDIC option is useless for a URL that was built on an ASCII system and then translated to EBCDIC by, for example, a mail gateway since the escape sequences will contain the ASCII value of the characters.

uro—Write Unit Record Output

uro copies the lines in the pipeline to a virtual unit record output device (printer or punch).



Type: Arcane device driver.

Placement: *uro* must not be a first stage.

Syntax Description: Arguments are optional. Specify the device address of the virtual printer or punch to write to if it is not the default 00E. The virtual device must be a unit record output printer or punch device. The keyword STOP allows you to inspect the channel programs built by *uro*.

Operation: The first byte of each record designates the CCW command code (machine carriage control character); it is inserted as the CCW command code. The remaining characters are identified for transport to SPOOL by the address and length fields of the CCW. A single blank character is written if the input record has only the command code. Control and no operation CCWs can specify data; the data are written to the SPOOL file. X'5A' operation codes are supported, but other read commands are rejected with an error message; command codes are not otherwise inspected.

Records may be buffered by *uro* to improve performance by writing more than one record with a single call to the host interface. A null input record causes *uro* to flush the contents of the buffer into SPOOL, but the null record itself is not written to SPOOL. After the producing stage has written a null record, it is assured that *uro* can close the unit record device without loss of data. Input lines are copied to the primary output stream, if it is connected.

uro issues no CP commands; specifically, the virtual device is not closed.

The virtual Forms Control Buffer (FCB) for a virtual printer (the virtual carriage control tape) can be loaded by a CCW or the CP command LOADVFCB. The channel program is restarted after a channel 9 or 12 hole causes it to terminate; even so, such holes in the carriage tape should be avoided, because they serve no useful purpose; and they generate additional overhead.

uro has not been tested with a dedicated printer or a dedicated punch.

Record Delay: *uro* strictly does not delay the record.

Commit Level: *uro* starts on commit level -2000000000. It ensures that the device is not already in use by another stage, allocates a buffer, and then commits to level 0.

See Also: *printmc*, *punch*, and *reader*.

Examples:

To close the printer every 50 records:

```
'PIPE (end ?)',
  '?... ',
  '|o: fanout',          /* Get two copies          */
  '|i:faninany',        /* Merge with nulls       */
  '|uro 00e',           /* Print; nulls flush     */
  '?o:',               /* The records            */
  '|chop 0',           /* Make them null         */
  '|join 49',          /* Join 50 null records   */
  '|c: fanout',        /* Still a null record    */
  '|i:',               /* Send to printer        */
  '?c:',               /* Trigger record         */
  '|spec /CLOSE 00E/', /* Build command          */
  '|cp'                /* Issue it                */
```

utf

The trick is to pass a null record to *uro* to force it to flush the contents of its buffer into CP SPOOL before the device is closed.

Notes:

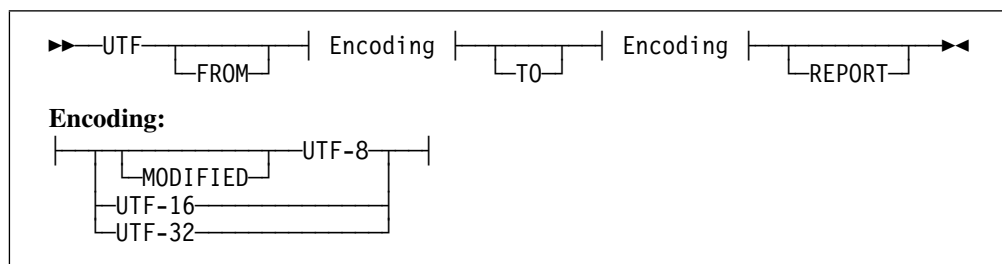
1. Use *punch* to write records without carriage control to a virtual punch.
2. Set NOPDATA on to write into SPOOL any data in a record that has a carriage control designating no operation (X'03').
3. Any output data can be written, including 3800 CCWs, but be aware that CP support depends on the virtual device type. For example, the maximum record length (including CCW operation code prefix) is 133 bytes on a virtual 1403.
4. STOP causes CP console function mode to be entered after each channel program has been given to CP. General register 2 points to the HCPSPGIOP data area, from which information about the channel program can be extracted.

Make sure you SET RUN OFF when using this option. This function was written to help debug *uro*, but it may also be useful to discover errors in input data.

***utf*—Convert between UTF-8, UTF-16, and UTF-32**

utf converts data encoded in any of the formats UTF-8, UTF-16, and UTF-32 to any other of these formats.

UTF-8 is variable length encoding of Unicode that has the property that 7-bit ASCII is encoded unchanged. UTF-16 is a fixed length encoding that is close to Unicode, but see the usage notes below. UTF-32 stores the Unicode code point in 32-bits.



Type: Filter.

Syntax Description:

- UTF-8 Variable length byte stream encoding that has the property that the first 128 values are 7-bit ASCII.
- MODIFIED UTF-8 UTF-8 encoding where U+0000 is encoded as two bytes (X'C080'). This has the advantage that the byte X'00' cannot legally occur in such a string.
- UTF-16 Halfword encoding where the assigned Unicode code points in the *Multi Lingual Plane* (MLP, U+0000 through U+FFFF) are encoded as the same value with the most significant byte first. A value larger than X'FFFF' (U+10000 through U+10FFFF) is encoded as a “surrogate pair”; that is, two halfwords using one code point in the range X'D800' through X'DBFF' followed by a code point in the range X'DC00' through X'DFFF' for a total of twenty bits.

: UTF-32 Fullword encoding containing the binary value of the Unicode code
 : point with the most significant byte first.

: REPORT Report input data that are not valid; that is, issue a message and termi-
 : nate. The default is to substitute U+FFFD for the code point(s) in error.

: ***Input Record Format:***

: *UTF-8:* This format uses from one to four bytes to encode the Unicode character set. It
 : offers many encodings that are not valid. In particular, *overlong encodings* are possible.
 : Such encodings use more bits than necessary to encode a Unicode code point. For
 : example, X'41' and X'C181' encode "A", but the second encoding is not valid.

: Valid encoded strings consists of byte strings of these formats:

- : • B'0xxxxxx'; 7-bit ASCII. X'00' is valid only when MODIFIED is omitted.
- : • B'110ppppx 10xxxxxx' encodes U+80 through U+7FF. The four p bits must not all
 : be zero, except that X'C080' encodes U+0000 when MODIFIED is specified.
- : • B'1110pppp 10pxxxxx 10xxxxxx' encodes U+800 through U+FFFF. The four p bits
 : must not all be zero, nor are code points in the range U+D800 through U+DFFF
 : allowed.
- : • B'11110ppp 10ppxxxx 10xxxxxx 10xxxxxx' encodes U+10000 through U+10FFFF.
 : The five p bits must not all be zero.

: *UTF-16:* This format uses two bytes to encode the valid code points in the MLP. Values
 : in the higher planes are encoded in a *surrogate pair*, which is four bytes of the form
 : B'110110pp ppxxxxxx 110111xx xxxxxxxx', where pppp is one less the number of the
 : plane (thus, a code point in the MLP cannot be encoded as a surrogate pair).

: *UTF-32:* The 22-bit code point number is stored in 32 bits. Values larger than
 : X'0010FFFF' are not valid.

: ***Streams Used:*** Records are read from the primary input stream and written to the primary
 : output stream. Null input records are discarded.

: ***Record Delay:*** *utf* does not delay the record.

: ***Premature Termination:*** *utf* terminates when it discovers that its output stream is not
 : connected.

: ***See Also:*** *xlate*.

: ***Notes:***

- : 1. In Unicode terminology, a *code point* represents an unsigned value in the range 0
 : through 1114111 (X'10FFFF'). A code point uniquely identifies a character or
 : control code.

: Unicode code points are by convention marked up as U+xxxx, where the value is
 : specified in hexadecimal.

- : 2. Use the same encoding format for input and output operands to validate an encoded
 : data stream without conversion.

var

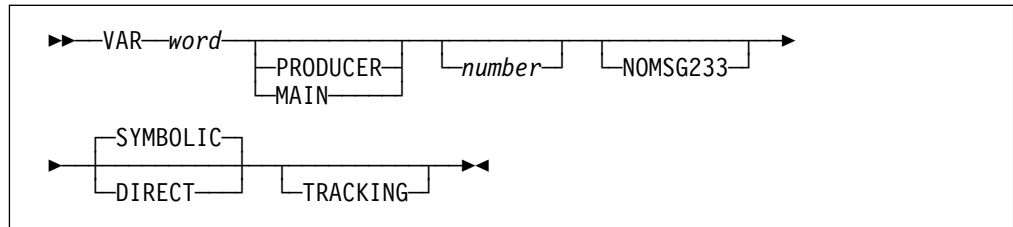
Publications:

As of this writing (January 2010), the current Unicode standard can be found at <http://www.unicode.org/versions/Unicode5.2.0/>

RFC 3629 describes UTF-8, but so does the current Unicode standard.

***var*—Retrieve or Set a Variable in a REXX or CLIST Variable Pool**

var connects a variable to the pipeline. When *var* is first in the pipeline, the contents of the specified variable are written to the pipeline. When *var* is not first in a pipeline, it sets the specified variable to the contents of the first input record and then passes all input to the output; the variable is dropped if there is no input and TRACKING is omitted.



Type: Device driver.

Warning: *var* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *var* is unintentionally run other than as a first stage. To use *var* to read data into the pipeline at a position that is not a first stage, specify *var* as the argument of an *append* or *preface* control. For example, `|append var ...|` appends the data produced by *var* to the data on the primary input stream.

Syntax Description: A word is required to specify the variable to fetch or store. It is possible to access a REXX variable pool other than the current one.

The keyword `PRODUCER` may be used when the pipeline specification is issued with `CALLPIPE`. It specifies that the variable pool to be accessed is the one for the stage that produces the input to the stage that issues the subroutine pipeline that contains *var*, rather than the current stage. (This is a somewhat esoteric option.) To ensure that the variable pool persists as long as this invocation of *var*, the stage that is connected to the currently selected input stream must be blocked in an `OUTPUT` pipeline command while the subroutine pipeline is running.

The keyword `MAIN` specifies that the REXX variable pool to be accessed is the one in effect at the time the pipeline set was created (either by the `PIPE` command or by the *runpipe* stage). `MAIN` is implied for pipelines that are issued with `ADDPPIPE`.

A number that is zero or positive is optional. It specifies the number of REXX variable pools to go back. That is, *var* can operate on variables in the program that issued the pipeline specification to invoke *var* or in one of its ancestors. (When the number is prefixed by either `PRODUCER` or `MAIN`, the variable pool to be accessed is the producer's or the main one, or one of their ancestors.) On CMS, if the number is larger than the number of REXX environments created on the call path from the `PIPE` command, *var* continues on the `SUBCOM` chain starting with the environment active when `PIPE` was issued.

Specify the option NOMSG233 to suppress message 233 when the REXX environment does not exist. Either way, *var* terminates with return code 233 on commit level -1 when the environment does not exist.

The keyword SYMBOLIC specifies that REXX should treat the variable names generated as it would a variable that is written in a program. DIRECT specifies that REXX should use the variable name exactly as written.

The keyword TRACKING specifies that *var* should continuously obtain the value of the variable and write it to the output or (if *var* is not first in a pipeline) set the variable to the contents of each input record, as it is read.

Operation:

When *var* is first in the pipeline, and TRACKING is omitted, *var* writes a single record containing the value of the variable and terminates.

When *var* is first in the pipeline, and TRACKING is specified, *var* continuously suspends itself and then writes the current value of the variable until it senses end-of-file on the primary output stream.

Note: Be sure that the pipeline limits the number of records consumed when *var* TRACKING is first in a pipeline; it does not terminate normally.

When *var* is not first in a pipeline, and TRACKING is omitted, it sets the variable from the first record read, passes the record to the primary output stream, consumes the record, and then shorts the primary input stream to the primary output stream. *var* drops the variable if no input record arrives.

When *var* is not first in a pipeline, and TRACKING is specified, it sets the variable as records become available and then passes the record to the primary output stream.

Record Delay: *var* strictly does not delay the record.

Commit Level: *var* starts on commit level -1. It verifies that the REXX environment exists (if it did not do so while processing its parameters) and then commits to level 0.

See Also: *stem* and *varload*.

Examples: To reverse the current line in the current XEDIT session, irrespective of its length.

```
/* REVCL XEDIT: Reverse current line */
'extract ,curline'
nuline=reverse(curline.3)
address command,
  'PIPE var nuline | xedit'
exit RC
```

XEDIT advances the current line pointer after a record is read or replaced; therefore, the EXTRACT XEDIT subcommand is used (rather than the *xedit* device driver) to get the contents of the current line.

The cascade of *split* and *drop* is useful to set several variables to different words in the input line:

```
pipe ... | split | var word1 | drop | var word2
```

var

split reformats the file to have a record for each blank-delimited word in the input. The first *var* sets the variable `word1` to the contents of the first line (which contains the first word of the input file), and then copies the input to the output. The *drop* stage discards the first record (which has already been stored); it passes the second word of the input file as the first record on the output. Thus, the first line that is read by the second *var* stage contains the second word of the input file. This word is then stored in the variable `word2`. Though you can add as many *drop-var* pairs as you like, it may be simpler to set a stemmed array when there are many words in the input file.

Note these three ways of using *var*. They produce the same result when the input file contains one record:

```
... | var x
... | var x tracking
... | append literal | var x
```

When there is no input file, the variable is dropped in the first example; the variable is left unchanged in the second example; and the variable is set to a null value in the third example (because *append* can always supply a null record).

When there is more than one input record, the first and third examples set the variable to the contents of the first record, but the second example sets it to the contents of the last record.

To set a variable to a Boolean indication of the presence of data:

```
... | stem x. | take 1 | count lines | var haveData
```

The file is stored in a stemmed array by *stem*. Then the first line is selected and counted. The count will be zero if there are no lines in the file. Because the count can only be zero or one, the variable `haveData` is set to a Boolean value.

Notes:

1. When a pipeline is issued as a TSO command, IKJCT441 is called to access the variable pool. When the command is issued with Address Link or Address Attach, *var* accesses the REXX environment from where the command is issued.
2. *CMS Pipelines* maintains a reference to the current variable environment for each stage. Initially this is the environment in effect for the PIPE command with which the original pipeline was started.

When a REXX program is invoked (as a stage or with the REXX pipeline command), its environment becomes the current one, with a pointer to the previous one.

When a pipeline specification is issued with the *runpipe* built-in program or the CALLPIPE pipeline command, the current environment is the one in effect for the stage issuing *runpipe* or CALLPIPE; it is known to persist while the subroutine pipeline runs. On the other hand, when a pipeline specification is issued with the ADDPIPE pipeline command, the stage that issues ADDPIPE runs in parallel with the added pipeline specification; it can terminate at any time (indeed, even before the new pipeline specification starts running). Therefore, for ADDPIPE, the current environment is set to the one for the last *runpipe* or the one at initial entry on the PIPE command. Thus, the MAIN option has effect only for pipeline specifications that are issued by the CALLPIPE pipeline command.

- Unless `DIRECT` is specified, `var` uses the symbolic interface to access REXX variables. This means that you should write the variable name the same way you would write it in an assignment statement. Consider this program fragment:

```
/* Process an array */
x='fred'
'PIPE literal a | var z.x'
```

The variable `Z.fred` is set to `'a'`. On the other hand, the following would set the variable `Z.x`:

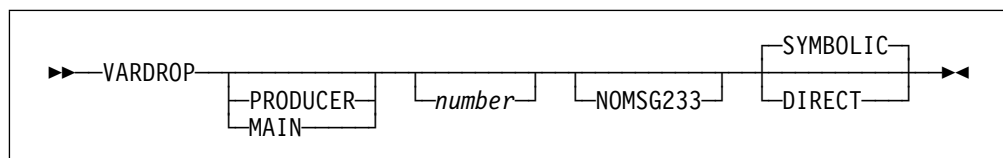
```
/* Process directly */
'PIPE literal a | var Z.x direct'
```

Note that the stem must be in upper case when `DIRECT` is used.

- An unset variable (that is, a variable that has been dropped or has never been assigned a value) is treated differently by the three variable repositories: REXX returns the name of the variable in upper case; EXEC2 and CLIST return the null string.
- Use `TRACKING` when you wish to leave the current value of the variable unchanged if there are no input records. Use `take 1` to ensure there is only one input record.

vardrop—Drop Variables in a REXX Variable Pool

vardrop reads input records that contain the names of variables to be dropped from a REXX variable pool.



Type: Device driver.

Placement: *vardrop* must not be a first stage.

Syntax Description: The arguments are optional. It is possible to access a REXX variable pool other than the current one.

The keyword `PRODUCER` may be used when the pipeline specification is issued with `CALLPIPE`. It specifies that the variable pool to be accessed is the one for the stage that produces the input to the stage that issues the subroutine pipeline that contains *vardrop*, rather than the current stage. (This is a somewhat esoteric option.) To ensure that the variable pool persists as long as this invocation of *vardrop*, the stage that is connected to the currently selected input stream must be blocked in an `OUTPUT` pipeline command while the subroutine pipeline is running.

The keyword `MAIN` specifies that the REXX variable pool to be accessed is the one in effect at the time the pipeline set was created (either by the `PIPE` command or by the *runpipe* stage). `MAIN` is implied for pipelines that are issued with `ADDPPIPE`.

A number that is zero or positive is optional. It specifies the number of REXX variable pools to go back. That is, *vardrop* can operate on variables in the program that issued the pipeline specification to invoke *vardrop* or in one of its ancestors. (When the number is prefixed by either `PRODUCER` or `MAIN`, the variable pool to be accessed is the producer's or the main one, or one of their ancestors.) On CMS, if the number is larger than the number

vardrop

of REXX environments created on the call path from the PIPE command, *vardrop* continues on the SUBCOM chain starting with the environment active when PIPE was issued.

Specify the option NOMSG233 to suppress message 233 when the REXX environment does not exist. Either way, *vardrop* terminates with return code 233 on commit level -1 when the environment does not exist.

The keyword SYMBOLIC specifies that REXX should treat the variable names generated as it would a variable that is written in a program. DIRECT specifies that REXX should use the variable name exactly as written.

Input Record Format: One variable per input record. The name of the variable begins in the first column of the record. Trailing blanks are retained.

Record Delay: *vardrop* strictly does not delay the record.

Commit Level: *vardrop* starts on commit level -1. It verifies that the REXX environment exists (if it did not do so while processing its parameters) and then commits to level 0.

See Also: *var*, *varfetch*, and *varset*.

Examples: To drop two variables in the EXEC that invoked the PIPE:

```
/* Now drop the variables */  
'callpipe literal oscar petrea | split | vardrop main'
```

Notes:

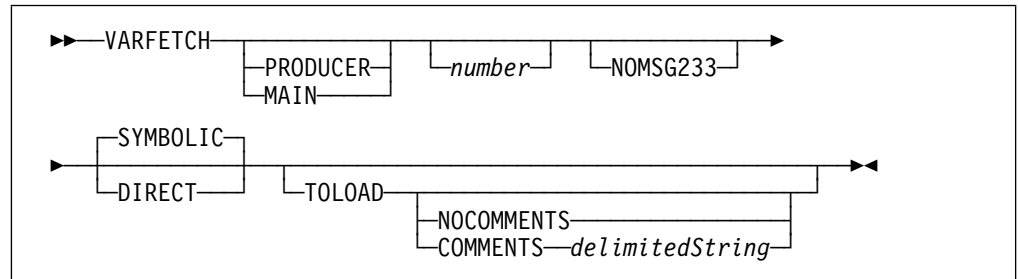
1. On z/OS, if *vardrop* is used in a pipeline specification that is issued with the PIPE command, the command must be issued by Address LINK.
2. *CMS Pipelines* maintains a reference to the current variable environment for each stage. Initially this is the environment in effect for the PIPE command with which the original pipeline was started.

When a REXX program is invoked (as a stage or with the REXX pipeline command), its environment becomes the current one, with a pointer to the previous one.

When a pipeline specification is issued with the *runpipe* built-in program or the CALLPIPE pipeline command, the current environment is the one in effect for the stage issuing *runpipe* or CALLPIPE; it is known to persist while the subroutine pipeline runs. On the other hand, when a pipeline specification is issued with the ADDPIPE pipeline command, the stage that issues ADDPIPE runs in parallel with the added pipeline specification; it can terminate at any time (indeed, even before the new pipeline specification starts running). Therefore, for ADDPIPE, the current environment is set to the one for the last *runpipe* or the one at initial entry on the PIPE command. Thus, the MAIN option has effect only for pipeline specifications that are issued by the CALLPIPE pipeline command.

varfetch—Fetch Variables in a REXX or CLIST Variable Pool

varfetch reads input records that contain the names of variables to be read from a variable pool. The value of the named variables is written to the output.



Type: Device driver.

Placement: *varfetch* must not be a first stage.

Syntax Description: An argument is optional. It is possible to access a REXX variable pool other than the current one.

The keyword `PRODUCER` may be used when the pipeline specification is issued with `CALLPIPE`. It specifies that the variable pool to be accessed is the one for the stage that produces the input to the stage that issues the subroutine pipeline that contains *varfetch*, rather than the current stage. (This is a somewhat esoteric option.) To ensure that the variable pool persists as long as this invocation of *varfetch*, the stage that is connected to the currently selected input stream must be blocked in an `OUTPUT` pipeline command while the subroutine pipeline is running.

The keyword `MAIN` specifies that the REXX variable pool to be accessed is the one in effect at the time the pipeline set was created (either by the `PIPE` command or by the *runpipe* stage). `MAIN` is implied for pipelines that are issued with `ADDPPIPE`.

A number that is zero or positive is optional. It specifies the number of REXX variable pools to go back. That is, *varfetch* can operate on variables in the program that issued the pipeline specification to invoke *varfetch* or in one of its ancestors. (When the number is prefixed by either `PRODUCER` or `MAIN`, the variable pool to be accessed is the producer's or the main one, or one of their ancestors.) On CMS, if the number is larger than the number of REXX environments created on the call path from the `PIPE` command, *varfetch* continues on the `SUBCOM` chain starting with the environment active when `PIPE` was issued.

Specify the option `NOMSG233` to suppress message 233 when the REXX environment does not exist. Either way, *varfetch* terminates with return code 233 on commit level -1 when the environment does not exist.

The keyword `SYMBOLIC` specifies that REXX should treat the variable names generated as it would a variable that is written in a program. `DIRECT` specifies that REXX should use the variable name exactly as written.

Specify `TOLOAD` to write output records in the format required as input to *varset* (and to *varload*): each record contain the variable's name as a delimited string followed by the variable's value. The delimiter is selected from the set of characters that do not occur in the name of the variable; it is unspecified how this delimiter is selected. The keyword `COMMENTS` is followed by a delimited string that enumerates the characters that should not

varfetch

be used as delimiter characters. The keyword `NOCOMMENTS` specifies that the delimiter character can be any character that is not in the variable's name. `NOCOMMENTS` is the default.

Operation: When the secondary output stream is not defined, *varfetch* writes an output record to the primary output stream for each input record. This record contains the value returned from the environment.

When the secondary output stream is defined, *varfetch* inspects the `SHVNEWV` flag to see if the variable exists. If the flag indicates that the variable does not exist, the input record is copied to the secondary output stream. If the flag indicates that the variable does exist, an output record is built and written to the primary output stream.

Input Record Format: One variable per input record. The name of the variable begins in the first column of the record. Trailing blanks are retained.

Output Record Format: When `TOLOAD` is omitted, the output record contains the value of the variable.

When `TOLOAD` is specified, output records are produced that can be used in *varset*. The output record contains:

1. A delimiter character (in column 1).
2. A variable's name (beginning in column 2).
3. A delimiter character (a copy of the character in column 1).
4. The variable's value.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null input records are discarded.

Record Delay: *varfetch* does not delay the record.

Commit Level: *varfetch* starts on commit level -1. It verifies that the REXX environment exists (if it did not do so while processing its parameters) and then commits to level 0.

Premature Termination: *varfetch* terminates when it discovers that no output stream is connected.

See Also: *var*, *vardrop*, *varload*, and *varset*.

Examples: Obtain the value of two variables in the EXEC that invoked the PIPE:

```
/* Now get the variables into our environment */
'callpipe (name VARFETCH)',
'|literal oscar petrea',          /* the variable names      */
'|split ',                        /* One per line           */
'|varfetch main toload',         /* Get the variables      */
'|varset direct'                 /* Store locally          */
```

Notes:

1. *varfetch* is identical to *varload*, except for the defaults.
2. When a pipeline is issued as a TSO command, IKJCT441 is called to access the variable pool. When the command is issued with Address Link or Address Attach, *varfetch* accesses the REXX environment from where the command is issued.

3. *CMS Pipelines* maintains a reference to the current variable environment for each stage. Initially this is the environment in effect for the PIPE command with which the original pipeline was started.

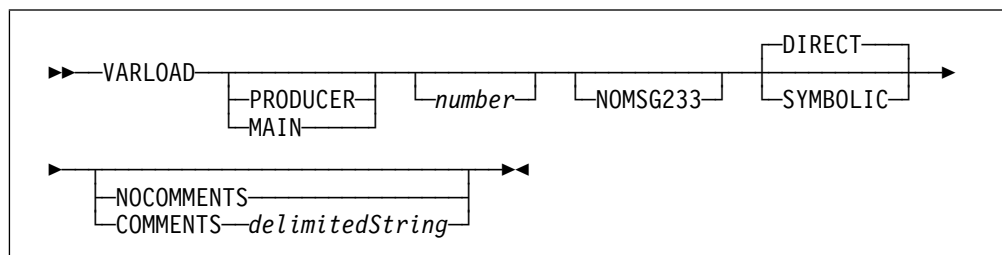
When a REXX program is invoked (as a stage or with the REXX pipeline command), its environment becomes the current one, with a pointer to the previous one.

When a pipeline specification is issued with the *runpipe* built-in program or the CALLPIPE pipeline command, the current environment is the one in effect for the stage issuing *runpipe* or CALLPIPE; it is known to persist while the subroutine pipeline runs. On the other hand, when a pipeline specification is issued with the ADDPIPE pipeline command, the stage that issues ADDPIPE runs in parallel with the added pipeline specification; it can terminate at any time (indeed, even before the new pipeline specification starts running). Therefore, for ADDPIPE, the current environment is set to the one for the last *runpipe* or the one at initial entry on the PIPE command. Thus, the MAIN option has effect only for pipeline specifications that are issued by the CALLPIPE pipeline command.

4. An unset variable (that is, a variable that has been dropped or has never been assigned a value) is treated differently by the three variable repositories: REXX returns the name of the variable in upper case; EXEC2 and CLIST return the null string. Only REXX sets the SHVNEWV flag.

varload—Set Variables in a REXX or CLIST Variable Pool

varload sets the values of variables based on the contents of its input records.



Type: Device driver.

Placement: *varload* must not be a first stage.

Syntax Description: It is possible to access a REXX variable pool other than the current one.

The keyword PRODUCER may be used when the pipeline specification is issued with CALLPIPE. It specifies that the variable pool to be accessed is the one for the stage that produces the input to the stage that issues the subroutine pipeline that contains *varload*, rather than the current stage. (This is a somewhat esoteric option.) To ensure that the variable pool persists as long as this invocation of *varload*, the stage that is connected to the currently selected input stream must be blocked in an OUTPUT pipeline command while the subroutine pipeline is running.

The keyword MAIN specifies that the REXX variable pool to be accessed is the one in effect at the time the pipeline set was created (either by the PIPE command or by the *runpipe* stage). MAIN is implied for pipelines that are issued with ADDPIPE.

varload

A number that is zero or positive is optional. It specifies the number of REXX variable pools to go back. That is, *varload* can operate on variables in the program that issued the pipeline specification to invoke *varload* or in one of its ancestors. (When the number is prefixed by either PRODUCER or MAIN, the variable pool to be accessed is the producer's or the main one, or one of their ancestors.) On CMS, if the number is larger than the number of REXX environments created on the call path from the PIPE command, *varload* continues on the SUBCOM chain starting with the environment active when PIPE was issued.

Specify the option NOMSG233 to suppress message 233 when the REXX environment does not exist. Either way, *varload* terminates with return code 233 on commit level -1 when the environment does not exist.

The keyword SYMBOLIC specifies that REXX should treat the variable names generated as it would a variable that is written in a program. DIRECT specifies that REXX should use the variable name exactly as written. The keyword COMMENTS is followed by a delimited string that enumerates the characters that can mark comment lines in the input. The keyword NOCOMMENTS specifies that the input contains no comment records. The default is COMMENT /* /.

Input Record Format: Records that contain one of the characters in the comment string in the first column are considered comments and are ignored. The first position of each is a delimiter character unless the record is treated as a comment. The name of the variable to set begins in column 2 and ends at the next occurrence of the delimiter character. That is, a *delimitedString* beginning in column 1 defines the name of the variable to set. In order that stemmed variables with any stem can be loaded, the variable name is not translated in any way; simple variables (and stems) must be in upper case. There is no substitution in stemmed variables when they are set.

Data to load into the variable, if any, immediately follow the second occurrence of the delimiter character and extend to the end of the record; use *strip* TRAILING to remove trailing blanks.

Record Delay: *varload* strictly does not delay the record.

Commit Level: *varload* starts on commit level -1. It verifies that the REXX environment exists (if it did not do so while processing its parameters) and then commits to level 0.

See Also: *stem*, *var*, and *varset*.

Examples: To store the CP settings in an associative array:

```
/* Store CP SETs */
'PIPE' ,
  'CP QUERY SET',          /* Do the CP command      */
  '| split ','',          /* One option per line    */
  '| strip',              /* Remove blanks          */
  '| spec /=CPSET./ next', /* Set stem name          */
      'word 1 next',      /* ... index              */
      '/=/ next',        /* Terminator             */
      'word 2-* next',   /* Value                  */
  '| varload'             /* Load variables        */
```

```
msg='MSG'; vmconio='VMCONIO'
Say cpset.msg cpset.vmconio
```

To load variables defining the GDDM codes for colours:

```

/* Setvars subroutine to set variables */
address Command,
  'PIPE < gddm setvars|varload 1'
exit RC

```

A sample input file for this is shown here; note that the variable names are all in upper case.

```

* GDDM SETVARS:
,WHITE,-2
,BLACK,-1
,BLUE,1
,RED,2
,MAGENTA,3
,GREEN,4
,CYAN,5
,YELLOW,6
,NEUTRAL,7
,BACKGROUND,8

```

rexxvars can be used to save the variables in a program so that they can be restored later:

```

/* Save variables */
'PIPE (name VARLOAD)',
  '| rexxvars', /* Read all variables */
  '| drop 1', /* Drop source string */
  '| spec /=/ 1 3-* next', /* Beginning of delimiter */
  '| join 1', /* Join name and value */
  '| > saved variables a' /* Write file */

/* Restore variables */
'PIPE (name VARLOAD)',
  '| < saved variables', /* Read variables */
  '| varload' /* Set them */

```

This example works with “well behaved” variables, but note that *rexxvars* cannot obtain the default value for a stem; it also truncates the value of a variable after 512 bytes. A more subtle thing to beware of is that a compound variable can have an equal sign as part of its name; this would cause a longer value to be restored than was saved.

To set all possible compound values whose names begin with STEM. (the default):

```
'PIPE literal /STEM./Value for array|varload'
```

Notes:

1. *varload* is identical to *varset*, except for the defaults.
2. When a pipeline is issued as a TSO command, IKJCT441 is called to access the variable pool. When the command is issued with Address Link or Address Attach, *varload* accesses the REXX environment from where the command is issued.
3. *CMS Pipelines* maintains a reference to the current variable environment for each stage. Initially this is the environment in effect for the PIPE command with which the original pipeline was started.

When a REXX program is invoked (as a stage or with the REXX pipeline command), its environment becomes the current one, with a pointer to the previous one.

When a pipeline specification is issued with the *runpipe* built-in program or the CALLPIPE pipeline command, the current environment is the one in effect for the stage

varset

issuing *runpipe* or *CALLPIPE*; it is known to persist while the subroutine pipeline runs. On the other hand, when a pipeline specification is issued with the *ADDPIPE* pipeline command, the stage that issues *ADDPIPE* runs in parallel with the added pipeline specification; it can terminate at any time (indeed, even before the new pipeline specification starts running). Therefore, for *ADDPIPE*, the current environment is set to the one for the last *runpipe* or the one at initial entry on the *PIPE* command. Thus, the *MAIN* option has effect only for pipeline specifications that are issued by the *CALLPIPE* pipeline command.

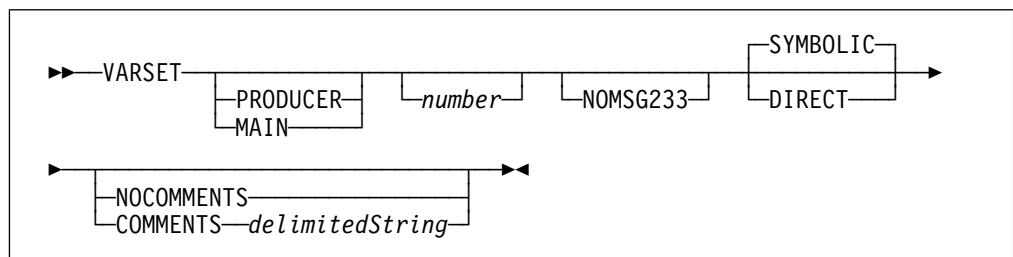
4. *varload* cannot set a compound variable whose derived name is the same as its stem (a compound variable with a null index). This can be accomplished with *var* if there exists a simple variable containing a null value; this example hijacks the question mark:

```
... | literal | var ? | drop 1 | var stem.?
```

A null record is generated by *literal*, stored in the question mark variable, and discarded. The next record is assigned to the compound variable with the null index.

varset—Set Variables in a REXX or CLIST Variable Pool

varset sets the values of variables based on the contents of its input records.



Type: Device driver.

Placement: *varset* must not be a first stage.

Syntax Description: An argument is optional. It is possible to access a REXX variable pool other than the current one.

The keyword *PRODUCER* may be used when the pipeline specification is issued with *CALLPIPE*. It specifies that the variable pool to be accessed is the one for the stage that produces the input to the stage that issues the subroutine pipeline that contains *varset*, rather than the current stage. (This is a somewhat esoteric option.) To ensure that the variable pool persists as long as this invocation of *varset*, the stage that is connected to the currently selected input stream must be blocked in an *OUTPUT* pipeline command while the subroutine pipeline is running.

The keyword *MAIN* specifies that the REXX variable pool to be accessed is the one in effect at the time the pipeline set was created (either by the *PIPE* command or by the *runpipe* stage). *MAIN* is implied for pipelines that are issued with *ADDPIPE*.

A number that is zero or positive is optional. It specifies the number of REXX variable pools to go back. That is, *varset* can operate on variables in the program that issued the pipeline specification to invoke *varset* or in one of its ancestors. (When the number is prefixed by either *PRODUCER* or *MAIN*, the variable pool to be accessed is the producer's or the main one, or one of their ancestors.) On CMS, if the number is larger than the number

of REXX environments created on the call path from the PIPE command, *varset* continues on the SUBCOM chain starting with the environment active when PIPE was issued.

Specify the option NOMSG233 to suppress message 233 when the REXX environment does not exist. Either way, *varset* terminates with return code 233 on commit level -1 when the environment does not exist.

The keyword SYMBOLIC specifies that REXX should treat the variable names generated as it would a variable that is written in a program. DIRECT specifies that REXX should use the variable name exactly as written. The keyword COMMENTS is followed by a delimited string that enumerates the characters that can mark comment lines in the input. The keyword NOCOMMENTS specifies that the input contains no comment records. The default is NOCOMMENT.

Operation: When the secondary output stream is not defined, *varset* copies the input record to the primary output stream after the variable is set.

When the secondary output stream is defined, *varset* inspects the SHVNEWV flag to see if the variable existed before. If the flag indicates that the variable already exists, the input record is copied to the primary output stream. If the variable is new, the input record is copied to the secondary output stream.

Input Record Format: Records that contain one of the characters in the comment string in the first column are considered comments and are ignored. The first position of each is a delimiter character unless the record is treated as a comment. The name of the variable to set begins in column 2 and ends at the next occurrence of the delimiter character. That is, a *delimitedString* beginning in column 1 defines the name of the variable to set. In order that stemmed variables with any stem can be loaded, the variable name is not translated in any way; simple variables (and stems) must be in upper case. There is no substitution in stemmed variables when they are set.

Data to load into the variable, if any, immediately follow the second occurrence of the delimiter character and extend to the end of the record; use *strip* TRAILING to remove trailing blanks.

Streams Used: Records are read from the primary input stream; no other input stream may be connected. Null input records are discarded.

Record Delay: *varset* strictly does not delay the record.

Commit Level: *varset* starts on commit level -1. It verifies that the REXX environment exists (if it did not do so while processing its parameters) and then commits to level 0.

Premature Termination: *varset* terminates when it discovers that no output stream is connected.

See Also: *var*, *vardrop*, and *varfetch*.

Notes:

1. When a pipeline is issued as a TSO command, IKJCT441 is called to access the variable pool. When the command is issued with Address Link or Address Attach, *varset* accesses the REXX environment from where the command is issued.

vchar

2. *CMS Pipelines* maintains a reference to the current variable environment for each stage. Initially this is the environment in effect for the PIPE command with which the original pipeline was started.

When a REXX program is invoked (as a stage or with the REXX pipeline command), its environment becomes the current one, with a pointer to the previous one.

When a pipeline specification is issued with the *runpipe* built-in program or the CALLPIPE pipeline command, the current environment is the one in effect for the stage issuing *runpipe* or CALLPIPE; it is known to persist while the subroutine pipeline runs. On the other hand, when a pipeline specification is issued with the ADDPIPE pipeline command, the stage that issues ADDPIPE runs in parallel with the added pipeline specification; it can terminate at any time (indeed, even before the new pipeline specification starts running). Therefore, for ADDPIPE, the current environment is set to the one for the last *runpipe* or the one at initial entry on the PIPE command. Thus, the MAIN option has effect only for pipeline specifications that are issued by the CALLPIPE pipeline command.

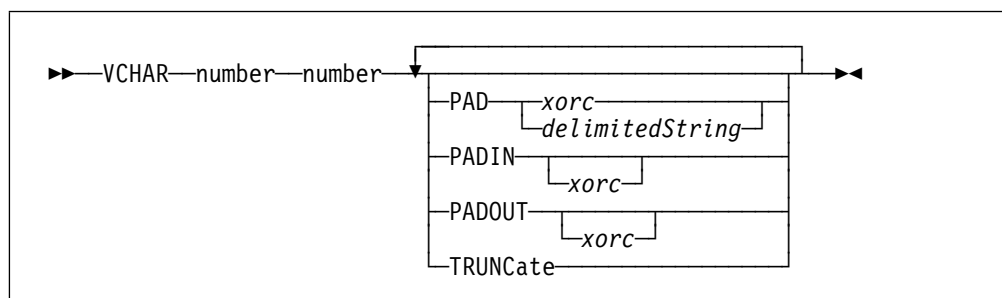
3. *varset* cannot set a compound variable whose derived name is the same as its stem (a compound variable with a null index). This can be accomplished with *var* if there exists a simple variable containing a null value; this example hijacks the question mark:

```
... | literal | var ? | drop 1 | var stem.?
```

A null record is generated by *literal*, stored in the question mark variable, and discarded. The next record is assigned to the compound variable with the null index.

vchar—Recode Characters to Different Length

vchar changes the character length, inserting or discarding leftmost bits. Input and output records are considered to contain characters of the length specified, spanned over bytes.



Type: Filter.

Syntax Description: The first argument is the number of bits per character in the input record; the second argument is the number of bits per character in the output record. Up to three options may follow the two numbers.

:	PAD	Padding to be inserted in each character when the output character is longer than the input one. Specify a single character or a delimited string. The character or string is replicated as much as required; bits from the left of this string are inserted in the leftmost added bits.
:	PADIN	Padding to be inserted at the end of each input record to complete the last character. This padding is at the right of the character. Specify a <i>xorc</i> or take the default padding of binary zeros.
:	PADOUT	Padding to be inserted at the end of each output record to complete the last byte. As many of the leftmost bits of the pad character as required are appended to the right of the last character. Specify a <i>xorc</i> or take the default padding of binary zeros.
:	TRUNCATE	Do not pad the output record; truncate any partial byte.

The default is PAD 00 PADOUT 00.

Operation: Bits are truncated on the left when the first number is larger than the second one. Zero bits are inserted on the left when the second number is larger than the first number. The input and output records are bit streams. A record is written for each input record. Only complete input characters are copied, effectively truncating the input record if it contains a number of bits that is not evenly divisible by the first number unless PADIN is specified. If the output record contains a number of bits that is not evenly divisible by eight, the last byte of the output record is padded with zeros on the right.

Record Delay: *vchar* strictly does not delay the record.

Premature Termination: *vchar* terminates when it discovers that its output stream is not connected.

Examples: To recode a file containing four 6-bit characters packed into every three 8-bit bytes:

```
...| vchar 6 8 |...
```

This example keeps the six bits together, adding two zero bits to the left of each byte.

To convert ASCII from 6-bit code to 8-bit code with three input bits and a leading zero into each output nibble (halfbyte):

```
...| vchar 3 4 |...
```

To truncate the record to contain an even number of bytes:

```
pipe literal a bb ccc dddd eeeee | split | vchar 16 16 | console
▶
▶bb
▶cc
▶dddd
▶eeee
▶Ready;
```

This exploits the fact that *vchar* discards any partial output field at the end of the input record. This can be modified to pad with an asterisk to an even length:

verify

```
pipe literal a bb ccc | split | spec 1-* 1 /*/ next | vchar 16 16 | ...
... console
▶a*
▶bb
▶ccc*
▶Ready;
```

First an asterisk is suffixed to each record; and then it is removed from records that contain an odd number of characters.

To insert a blank in front of each four characters (assuming that no input record contains X'00'):

```
pipe literal abcdefgh | vchar 32 40 | xlate *-* 00 blank | console
▶ abcd efgh
▶Ready;
```

For production use, this should be enhanced to suffix three blanks to the record before the *vchar* stage; otherwise the records are truncated when they contain a number of characters that is not evenly divisible by four. Using *pad* with MODULO and the PADIN on *vchar* avoids the *xlate* stage and the restriction on the X'00' characters.

```
pipe literal abcdefghi | pad modulo 4 | vchar 32 40 padin 40 | console
▶ abcd efgh i
▶Ready;
```

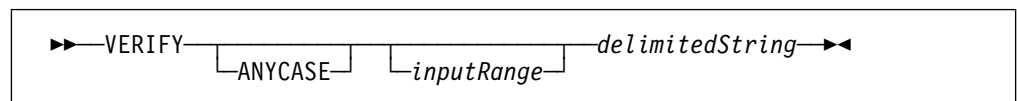
To write a string in two records, each pair of input characters arranged vertically:

```
pipe literal 47F0F120 | spec 1-* 2 write 1-* 1 | vchar 16 8 | console
▶4FF2
▶7010
▶Ready;
```

This example uses *spec* to produce two copies of the input record where the first one is offset one byte to the right. For each pair of input characters, *vchar* selects the rightmost character.

verify—Verify that Record Contains only Specified Characters

verify selects records that contain only specified characters. It rejects records that contain characters that are not present in the specified string.



Type: Selection stage.

Syntax Description: Specify ANYCASE to make the comparison case insensitive. An input range is optional. This specifies the part of the record to be inspected. The delimited string enumerates the characters that are allowed within the input range.

Operation: *verify* tests the characters within the input range for being in the specified string. If the input range is null or all characters in the range are in the specified string, the record is passed to the primary output stream. Otherwise, the record is discarded, or passed to the secondary output stream if the secondary output stream is connected.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *verify* strictly does not delay the record.

Commit Level: *verify* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *verify* terminates when it discovers that no output stream is connected.

Examples: To verify that a range contains a number:

```
... | verify 5.5 /0123456789/ | ...
```

To verify that a range contains at least one character that is not numeric:

```
... | not verify 5.5 /0123456789/ | ...
```

Notes:

1. *verify* is similar to the REXX built-in function `verify()`.
2. CASEANY, CASEIGNORE, CASELESS, and IGNORECASE are all synonyms for ANYCASE.

vmc—Write VMCF Reply

vmc sends commands to a service machine and writes the reply to the output. It uses the protocol used by the Realtime Monitors for VM/System Product, VM/XA* System Product, VM/ESA and z/VM.



Type: Device driver.

Syntax Description: The first word specifies the virtual machine to send commands to. An initial message is optional after the name of the virtual machine.

Operation: If an argument string is present, it is issued as the first message. Further messages are read from the input.

vmc sends messages over the Virtual Machine Communications Facility (VMCF) to a service machine. It expects a single reply. The reply is deblocked to 80-byte records and written to the output.

Commands must be 256 characters or shorter.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null and blank input records are discarded.

Record Delay: *vmc* writes all output for an input record before consuming the input record.

Premature Termination: *vmc* terminates when it discovers that its output stream is not connected.

Examples: To display part of a performance display of the PERFSVM service machine:

```
pipe vmc perfsvm 1 | take 7 | chop 62 | console
▶ FCX325      CPU 1090  SER 25F8B      CPU data menu
▶
▶
▶ CPU activity reports
▶ S Command   Description
▶ _ CPU       CPU Load and Transactions
▶ _ DSVBKACT  Dispatch Vector Activity
▶Ready;
```

Notes:

1. *vmc* interoperates with *vmclisten* and *vmcreply*, but as it deblocks the reply into 80-byte output records, *vmclient* may be a more appropriate choice.
2. IBM Performance Toolkit requires DATA authorisation for the user in FCONRMT AUTHORIZ to use the *vmc* to retrieve data through VMCF.
3. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (|) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

vmcdata—Receive, Reply, or Reject a Send or Send/receive Request

vmcdata can receive data explicitly, which may be required when RECEIVE is omitted from *vmclisten*. It completes a send/receive transaction by either rejecting or replying.



Type: Device driver.

Operation: The supported function codes are:

- | | |
|----------|---|
| VMCPRECV | Receive, X'0005'. The input record contains the message header only. The output record contains the data received appended to the parameter list. |
| VMCPREPL | Reply, X'0007'. The reply data must be appended to the message header in the input record. The reply function is also performed when the function code is unchanged from the message header (VMCPSNDR). |
| VMCPRJCT | Reject, X'000B'. |

Input Record Format: A 40-byte message header followed by optional reply data. The fields VMCMID and VMCMUSER must remain unchanged from *vmclisten*, as they identify the message being responded to.

Output Record Format: The 40-byte parameter list after the VMCF function has completed. For the receive function, data received are appended to the parameter list.

Record Delay: *vmcdata* does not delay the record.

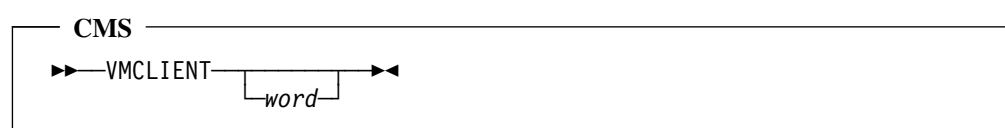
See Also: *vmclient* and *vmclisten*.

Publications:

z/VM CP Programming Services, SC24-6272, appendix C.

vmclient—Send VMCF Requests

vmclient sends requests to a virtual machine and writes the response to the primary output stream.



Type: Device driver.

Syntax Description: The *word* specifies the target virtual machine. When present, it is inserted in all input records.

Operation: *vmclient* issues the VMCF diagnose for each input record. For identify it then outputs the parameter list. For other functions, it waits for the final response interrupt, which indicates that the transaction is complete, and then produces an output record that contains the message header and reply data, if any.

Input Record Format: A VMCF parameter list (40 bytes) followed by data to transmit. The parameter list must have been filled in for function, user (unless an operand is specified), and (for send/receive) the length of the desired response buffer. The message identifier is reserved for *CMS Pipelines* use unless the identify function is specified.

Supported function codes are *send*, *sendx*, *send/receive*, and *identify*. The first buffer and length (VMCPVADA and VMCPLENA) are set to reflect the balance of the record from position 41 and on. For a *send/receive* function, a sufficient buffer is allocated for the response, as specified by VMCPLENB. If VMCPLENB is zero, the current size of the response buffer, which is at least 4056 bytes, is used.

Output Record Format: For *identify*, the parameter list; otherwise the message header (40 bytes) for the response interrupt followed by reply data, if any.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *vmclient* does not delay the record. That is, the output record is produced before the input record is consumed; however, there may well be a temporal delay while the server processes the request. *vmclient* waits forever if the server neither rejects the message nor produces a response.

Commit Level: *vmclient* starts on commit level -2. It ensures that the external interrupt infrastructure is available, that the virtual machine is authorized for VMCF, and then commits to level 0.

vmclisten

Premature Termination: *vmclient* terminates when it discovers that its output stream is not connected; *vmclient* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *vmclisten* and *vmcdata*.

Notes:

1. When an operand is specified and no *vmclisten* stage is active, a specific authorize is used for the target unless another *vmclient* stage is active and has specified a different user; the authorization is upgraded to full in this case, as it will be by *vmclisten*.

Publications:

z/VM CP Programming Services, SC24-6272, appendix C.

vmclisten—Listen for VMCF Requests

vmclisten listens for VMCF requests and writes the message header and possibly send data to the primary output stream for each VMCF interrupt it receives.



Type: Device driver.

Placement: *vmclisten* must be a first stage.

Syntax Description:

RECEIVE An automatic immediate receive is performed for messages indicating the send or the send/receive function. Thus, such messages cannot be rejected for the send function.

Output Record Format: The 40-byte message header followed by any sendx data or send data if RECEIVE is specified.

Commit Level: *vmclisten* starts on commit level -2. It ensures that the external interrupt infrastructure is available, that the virtual machine is authorized for VMCF, and then commits to level 0.

Premature Termination: *vmclisten* terminates when it discovers that its output stream is not connected; *vmclisten* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

See Also: *vmclient* and *vmcdata*.

Notes:

1. There can be at most one *vmclisten* stage active within a virtual machine at any one time.
2. If a specific authorize is active when *vmclisten* starts, it is upgraded to a general one.

3. When send/receive is indicated in the function code, the pipeline must generate an appropriate reply or reject and pass this to *vmcdata*.

Publications:

z/VM CP Programming Services, SC24-6272, appendix C.

waitdev—Wait for an Interrupt from a Device

waitdev waits for the next interrupt from a device and writes the subchannel status word to the primary output stream. When a real device is already present for the virtual device, *waitdev* shorts the primary input stream to the primary output stream and terminates immediately.

►►—WAITDEV—*devaddr*—◄◄

Type: Device driver.

Syntax Description:

devaddr The virtual device number of the device to wait on. The virtual device type must be terminal, graphic, unit record input, or channel to channel adapter.

Operation: *waitdev* may terminate for one of three reasons:

1. A real device is already present for the virtual device. This means that a terminal is already present. This cannot occur for readers and channel to channel adapters.

The primary input stream is shorted to the primary output stream.

2. An interrupt arrives on the virtual device. For terminals and graphics, it is likely a device end indicating that a user has dialled in. For readers, a device end indicates that a file has arrived. For channel to channel adapters, an attention indicates that the other side has made a channel command pending.

The 12 byte channel status word is written to the primary output stream.

3. The pipeline is signalled to stop.

No output is generated.

Record Delay: *waitdev* does not delay the record.

Commit Level: *waitdev* starts on commit level -2000000000. It verifies that the virtual device exists and is of a supported type, and then commits to level 0.

Premature Termination: *waitdev* terminates when it discovers that its output stream is not connected; *waitdev* also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

Examples: To wait for a user to dial in and then display the input 3270 data stream until the user generates an attention interrupt:

warp

```
: /* Wait for someone to dial in and then pass data to fullscr. */
: parse arg dev
: 'callpipe (end \ name WDIAL.REXX:10)',
:   '\literal',
:   '|waitdev' dev,
:   '|stem how.',
:   '|append literal', /* Be sure to shut the gate */
:   '|g:gate',
:   '\*:',
:   '|g:',
:   '|hole'
:
: If how.0=0
:   Then exit /* Stopped. */
: say dev 'now dialed.'
:
: 'callpipe (end \ name WDIAL.REXX:6)',
:   '\*:',
:   '|fullscr' dev 'asyn'
:
: address command 'CP RESET' dev
```

The first pipeline drains the input while waiting for the interrupt.

The second pipeline passes the input to the screen. Should the user generate an attention interrupt, *fullscr* will produce a record which will cause it to terminate as there is no consumer.


Notes:

waitdev does not inspect the interrupt status.

warp—Pipeline Wormhole

warp passes data through a wormhole from a *pitcher*, which is a *warp* stage that is not a first stage, to a *catcher*, which is a *warp* stage that is a first stage; the records are passed to the pitcher's primary output stream and also emanate from the catcher's primary output stream.

Within a pipeline set, there can be any number of pitcher stages, but at most one catcher stage by a particular name.



Type: Arcane gateway.

Syntax Description: Specify the wormhole's name as the only operand. The name is truncated after eight characters. Case is respected in wormhole names. The scope of a wormhole name is the pipeline set.

Operation: When *warp* is first in the pipeline, it waits for records to fall out of the wormhole and passes them to its primary output stream.

When *warp* is not first in the pipeline, it sends its input records through the wormhole. A pitcher will terminate without consuming the record if the catcher no longer exits or cannot

write the record. The pitcher waits for the catcher to complete writing its output record; it then passes the record to its own primary output stream, ignoring end-of-file.

Streams Used: Records are read from the primary input stream and written to the primary output stream. End-of-file is propagated from the catcher's primary output stream to the pitchers' primary input stream.

Record Delay: A pitcher *warp* stage delays the record until it has been written by the catcher.

Commit Level: *warp* starts on commit level -20. When it is first in the pipeline *warp* verifies that no other wormhole exist with the specified name and then commits to level 0. When it is not first in the pipeline, *warp* commits to level -1 to give a catcher time to start; if there then is no catcher, the stage terminates with an error message; otherwise it proceeds to commit level 0.

Premature Termination: A catcher terminates when it cannot write its output; this causes the pitcher to terminate as well.

A pitcher ignores end-of-file on its primary output stream.

warp also stops if the immediate command PIPMOD STOP is issued or if a record is passed to *pipestop*.

Examples: A rather contrived example:

```
pipe (end ?) literal howdy!|warp catch-22 ? warp catch-22 | console
▶howdy!
▶Ready;
```

Notes:

1. *warp* does not implement function that cannot be implemented with multistream pipelines, proper connection of streams, *faninany*, and, not least, sufficient stamina, but it is an easy way to gather data from a number of pipeline specifications.
2. *warp* sets the internal wait flag, which means that it cannot cause the pipeline set to go into a wait state; thus it cannot obscure a stall.
3. The workings of *warp* are similar to a UNIX named pipe. In fact, a pitcher can establish a catcher by a different name to receive a reply, and then send the private catcher's name in the message it pitches to a server that has a well known name.
However, such practice is not considered *pipethink*.
4. *warp* can send records back to a previously defined pipeline specification, notably one issued by ADDPIPE; but it cannot send records to a pipeline specification that has not yet been issued.

warplist—List Wormholes

warplist writes a record for each active wormhole in the pipeline set. The record contains the name of the wormhole, padded with blanks on the right to eight characters.

▶▶—WARPLIST—◀◀

Type: Arcane service program.

whilelabel

: **Placement:** *warplist* must be a first stage.

: **Premature Termination:** *warplist* terminates when it discovers that its output stream is

: not connected.

: **Examples:**

```
pipe (end ?) warp x|hole ? warp y|hole ? warplist | console
▶y
▶x
▶Ready;
```

whilelabel—Select Run of Records with Leading String

whilelabel selects input records up to the first one that does not begin with the specified string. That record and the records that follow are discarded.



Type: Selection stage.

Syntax Description: A string is optional. The string starts after exactly one blank character. Leading and trailing blanks are significant.

Operation: Characters at the beginning of each input record are compared with the argument string. Any record matches a null argument string. A record that is shorter than the argument string does not match.

whilelabel copies records up to (but not including) the first one that does not match to the primary output stream, or discards them if the primary output stream is not connected. *whilelabel* passes the remaining input records to the secondary output stream.

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected. *whilelabel* severs the primary output stream before it passes the remaining input records to the secondary output stream.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *whilelabel* strictly does not delay the record.

Commit Level: *whilelabel* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *whilelabel* terminates when it discovers that no output stream is connected.

See Also: *between*, *frolabel*, *inside*, *notinside*, *outside*, and *tolabel*.

Examples: To select the ESD cards from the first text deck in a file, discarding any update log in front of it:


```

/* FIRSTESD REXX */
'callpipe',
  '|*:',
  '|frlabel' '02'x || 'ESD',
  '|whilelabel' '02'x || 'ESD',
  '|*:'

```

To discard a run of records, all beginning with an asterisk:

```

/* DropComm REXX */
'callpipe (name WHILELAB)',
  '|*:',
  '|whilelabel *' ||,
  '|hole'

```

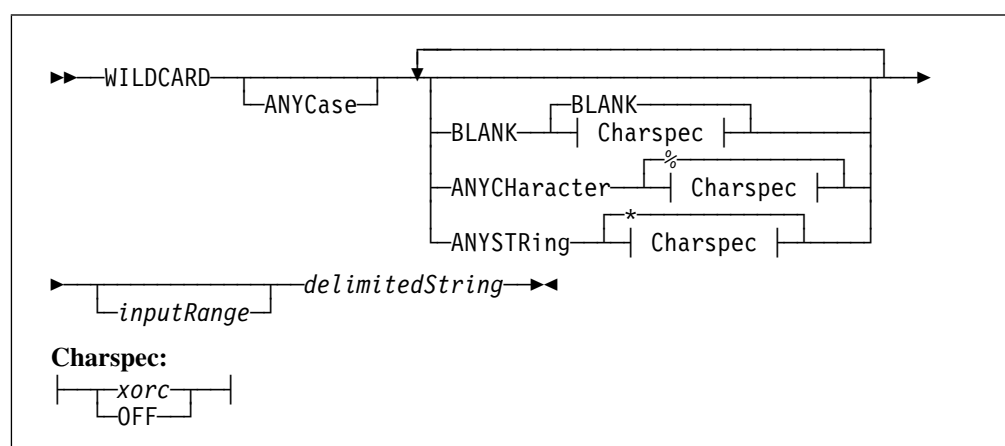
hole ensures that *whilelabel* does not terminate prematurely when it discovers that the primary output stream is not connected and there is no secondary output stream defined. The concatenation operator ensures that lines that contain just an asterisk are dropped, as are lines that have a character other than a blank in column 2.

Notes:

1. Use *strwhilelab* with ANYCASE for caseless compare.
2. *pick* can do what *whilelabel* does and much more.
3. Remember that REXX continuation functionally replaces a trailing comma with a blank. Also recall that when two strings are separated by one or more blanks, REXX concatenates them with a single blank. Use the concatenation operator (||) before the comma at the end of the line if your portrait style has the stage separators at the left side of the stage and the trailing blank is significant to your application.

wildcard—Select Records Matching a Pattern

wildcard is a selection stage that matches an input range in a way similar to the way in which the CMS command “LISTFILE” uses wildcards. *wildcard* selects records that match the specified string. It discards records that do not match the specified string.



Type: Selection stage.

Syntax Description:

ANYCASE	Comparison with literal characters is caseless.
BLANK	Specify the word delimiter. A string of one or more word delimiters in the pattern matches a string of one or more word delimiters in the input range. The default word delimiter is the blank.
ANYCHARACTER	Specify the pattern character that matches any single character in the input range, except word delimiters. Specify OFF to disable the any character. The default any character is the percent sign (%).
ANYSTRING	Specify the pattern character that matches zero or more characters in the input range, except for word delimiters. Specify OFF to disable the any string. The default any string is the asterisk (*).
<i>inputRange</i>	Specify the part of the record to be matched. The default is the entire record.
<i>delimitedString</i>	Specify the pattern to be matched against the input range.

The word delimiter, any character, and any string are collectively referred to as “meta characters”. The meta characters are case sensitive, irrespective of the case setting. This applies also to word delimiter characters in the input range.

The meta characters must all be different; this is enforced during the parse of the operands.

Meta characters may be defined more than once. The last occurrence of the definition of any particular meta character is the one used.

Operation: Matching the pattern against the input range is conceptually done by first breaking the pattern and the input range into words delimited by the word delimiter meta character. There must be the same number of words in both for the record to match. For each word, again conceptually, the literals, any characters and any strings in the pattern are matched against the word in the input range. A pattern word that consists of any strings only will match any single character, but not the empty string (as LISTFILE assumes that file names, types, and modes are not blank). In combination with the any character or literals, the any string can match the null string.

The handling of word delimiters at the boundaries of the input range is not symmetrical:

- In the left margin, they are observed rigorously. When the pattern includes no leading word delimiters, matching will fail when the input range contains a leading word delimiter; conversely, when the pattern includes a leading word delimiter, matching will fail when the input range contains no leading word delimiter.
- In the right margin, on the other hand, trailing word delimiters are allowed at the end of the input range. Thus, the pattern /a/ matches an input range that contains “a” in the first column and any number of trailing blanks. If the pattern contains a trailing word delimiter, the input range must have trailing word delimiter(s).

Streams Used: Records are read from the primary input stream. Secondary streams may be defined, but the secondary input stream must not be connected.

Record Delay: An input record is written to exactly one output stream when both output streams are connected. *wildcard* strictly does not delay the record.

Commit Level: *wildcard* starts on commit level -2. It verifies that the secondary input stream is not connected and then commits to level 0.

Premature Termination: *wildcard* terminates when it discovers that no output stream is connected.

Examples: To “swap” two meta characters, one of them must be disabled temporarily:

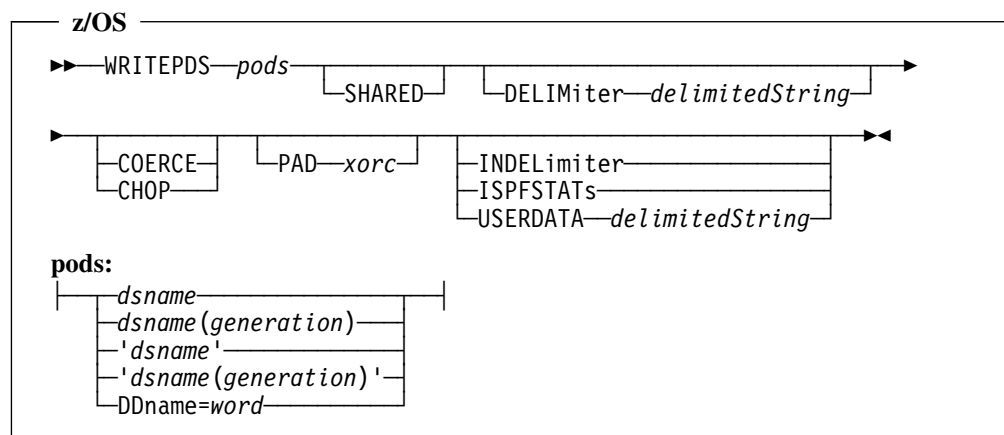
```
... | wildcard anychar off anystring % anychar * /*bcd%/ | ...
```

Notes:

1. When processing the output from the LISTFILE command that has more than three words, be sure to restrict the range to the first three words (usually 1.19). Also be sure to specify all three words, possibly using an any string meta character for the mode.

writepds—Store Members into a Partitioned Data Set

writepds stores members into a partitioned data set. The records for each member are prefixed by a record that specifies the member name.



Type: Device driver.

Placement: *writepds* must not be a first stage.

Syntax Description:

- pods** Enclose a fully qualified data set name in single quotes; the trailing quote is optional. Specify the DSNAMES without quotes to have the prefix, if any, applied. Append parentheses containing a signed number to specify a relative generation of a data set that is a member of a generation data group. To store members into an already allocated data set, specify the keyword DDNAME= followed by the DDNAME already allocated. The minimum abbreviation is DD=.
- SHARED** Allocate the data set shared rather than exclusive write. For a PDS you must ensure that no other stage or user allocates the data set for write concurrently.
- DELIMITER** Specify the delimiter string that separates members in the input stream. The string must match the leading characters of an input record. The following word of the input record is then taken to be the member name. The default is /*COPY /, which has a trailing blank.

INDELIMITER	User data to be stowed in the member is present in the delimiter record as an unpacked hexadecimal string, which follows the member name.
ISPFSTATS	Update or create status information associated with the member. This information is kept in the user data field of the PDS directory entry. The information is in the ISPF format.
USERDATA	Specify the user data to be associated with all members created or replaced. This information is kept in the user data field of the PDS directory entry. The data need not be in ISPF format.

The options COERCE, CHOP, and PAD are used with fixed record format data sets. COERCE specifies that the input records should be padded with blanks or truncated to the record length of the data set. CHOP specifies that long records are truncated; input records must be at least as long as the record length for the data set. PAD specifies the pad character to use when padding the record. Input records must not be longer than the record length of the data set when PAD is specified alone.

Input Record Format: The delimiter record contains the specified string beginning in column one. The member name is specified after the delimiter string. If it is requested, the following word contains the user data string. The first input record must be a delimiter (so that the name for the first member can be specified).

Streams Used: *writelnpds* passes the input to the output.

Record Delay: *writelnpds* strictly does not delay the record.

Commit Level: *writelnpds* starts on commit level -2000000000. It opens the DCB and then commits to level 0.

See Also: *listispf* and *listpds*.

Examples: To create the individual members of the *TSO Pipelines* help library:

```
/* Build Help library */
'PIPE',
  '|< dd=fplparms(fplhelp) ',
  '| unpack ',
  '| writelnpds dd=fplhelp delimiter /%COPY% /
```

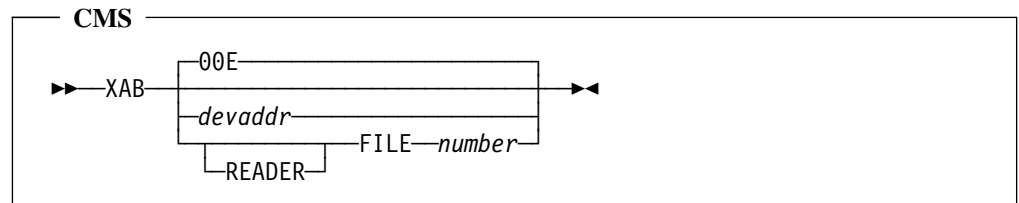
Members in the composite help file are separated by delimiter records that contain %COPY% in the left margin.

Notes:

1. *pdswrite* is a synonym for *writelnpds*.
2. Note that the delimiter is specified in a different way than done in *maclib*.

xab—Read or Write External Attribute Buffers

When *xab* is first in a pipeline it writes the contents of an external attribute buffer (XAB) into the pipeline; the buffer can be associated with a virtual printer device or a SPOOL file. When *xab* is not first in a pipeline, the first record in the pipeline is copied into the XAB for a virtual printer device or a SPOOL file.



Type: Device driver.

Warning: *xab* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *xab* is unintentionally run other than as a first stage. To use *xab* to read data into the pipeline at a position that is not a first stage, specify *xab* as the argument of an *append* or *preface* control. For example, `|append xab ...|` appends the data produced by *xab* to the data on the primary input stream.

Syntax Description: Arguments are optional. The default is to read or write the external attribute buffer of the virtual printer on address 00E. Specify a device address of a virtual printer to reach one at some other address. Write the SPOOL file number after the keyword FILE to process a particular file. Specify READER when the file is on the reader chain; the file is assumed to be on the printer chain if READER is omitted.

Operation: An external attribute buffer is read from a virtual printer or from a file when *xab* is first in a pipeline.

The buffer is replaced with the contents of the first input record when *xab* is not first in a pipeline. The input file is shorted to the output after the buffer is set successfully.

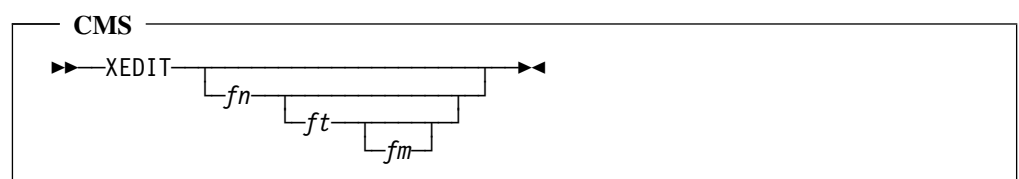
Record Delay: *xab* strictly does not delay the record.

Examples: To discover the XAB set up by Print Services Facility (PSF):

```
psf some file
pipe xab | > xab 00e a
```

xedit—Read or Write a File in the XEDIT Ring

xedit connects the pipeline to a file in the XEDIT ring; it reads lines from the file into the pipeline or writes lines from the pipeline into the file. Records written to the file replace existing records or are added to the end of the file.



Type: Device driver.

Warning: *xedit* behaves differently when it is a first stage and when it is not a first stage. Existing data can be overlaid when *xedit* is unintentionally run other than as a first stage. To use *xedit* to read data into the pipeline at a position that is not a first stage, specify *xedit* as the argument of an *append* or *preface* control. For example, `|append xedit ...|` appends the data produced by *xedit* to the data on the primary input stream.

Syntax Description: The arguments are the file name, type, and mode of the file to process. An asterisk (*) is used for components that are not specified. The default is the current file.

Operation: When *xedit* is first in the pipeline, lines from the file are copied into the pipeline starting at the current line pointer (which XEDIT advances to the next line after each line is read). *xedit* suspends itself to let other stages run before it obtains each record from the host interface. When *xedit* is resumed, it ensures that the primary output stream is still connected. The complete line up to the LRECL is available when reading from XEDIT; the VERIFY and ZONE settings have no effect. For variable record format files, XEDIT strips trailing blanks down to a minimum of one blank. Position the file at the top before reading to get all lines in the file.

When *xedit* is not first in a pipeline, lines in the file are replaced with records from the pipeline starting at the current line or are added to the file if the current line is at the end-of-file. (XEDIT advances to the next line after each line is replaced.) To replace the entire contents of a file, ensure that the file is empty before using *xedit* to add lines to the file. To add lines to the file, ensure that the current line is at the end-of-file. For fixed record format files, records must be exactly as long as the XEDIT logical record length; use *pad* to extend the record. You must have sufficient WIDTH for the longest record when you write to a variable record format file; lines are truncated at the width by XEDIT when SPAN is OFF. CASE, IMAGE, TABS, and TRUNC settings have no effect on lines appended to or replaced in the file by *xedit*. The record is also copied to the primary output stream (if it is connected).

Record Delay: *xedit* strictly does not delay the record.

Premature Termination: When it is first in a pipeline, *xedit* terminates when it discovers that its output stream is not connected.

See Also: *subcom* and *xmsg*.

Examples: Issue these commands from the XEDIT command line to count the number of blank-delimited words in the file (or you could make an XEDIT macro with the subcommands):

```
top
cms pipe xedit | count words | xmsg
```

The result is displayed as an XEDIT message. The file will be left positioned at the end of the file.

To count the number of words in the first five lines (or the entire file if it contains fewer than five lines):

```
:1 pipe xedit | take 5 | count words | xmsg
```

The file will be left positioned on line 6, because XEDIT moves the current line forward after *xedit* has obtained the fifth line.

To position the file after the first line that is longer than 80 bytes:

```
:0 pipe xedit | locate 81
```

This example relies on the fact that *locate* will terminate without consuming its current input record when it discovers that its output streams are no longer connected. As used here, *locate* will discard records that are 80 bytes or shorter without trying to write to an output stream. When *locate* writes the first line that is 81 bytes or longer, it discovers that its primary output stream is not connected and terminates. *xedit* will terminate because this severs its primary output stream. XEDIT will have advanced the read pointer to the next line.

Notes:

1. An XEDIT session must exist or be set up before PIPE is issued to process a pipeline specification. Queue or stack a pipeline command before invoking XEDIT:

```
/* Process reader files */
queue 'cms pipe cp q rdr * all | procrdr | xedit'
'XEDIT READER FILES S'
```

You can also issue the PIPE command from the XEDIT profile.

2. Lines are read and written in files in the topmost XEDIT ring; you cannot access files in an XEDIT session that has invoked XEDIT recursively.
3. You cannot directly insert records through the interface used by *xedit*; for records shorter than 253 bytes you might be able to use:

```
...| change //i / | Subcom Xedit...
```

Any lines inserted this way are processed according to the settings of TABS, IMAGE, CASE, and so on. You can also add a dummy line and overwrite it without having to worry about these settings:

```
'PIPE (end ? name XEDIT)',
  ' ...',
  '|o:fanout',                /* Make copy of file      */
  '|spec /command input /',  /* Insert a blank line    */
  '|subcom xedit',
  '|?o:',                    /* Second copy of line    */
  '|xedit',                  /* Overwrite current line */
  '|spec /command locate -1/', /* Command to back up    */
  '|subcom xedit'           /* Do so                  */
```

This works because all stages do not delay the record. For each input record, *fanout* first passes the record to its primary output stream where it is turned into an XEDIT command that adds a record to the file. XEDIT advances the current line pointer to point to the newly inserted line, which contain blanks.

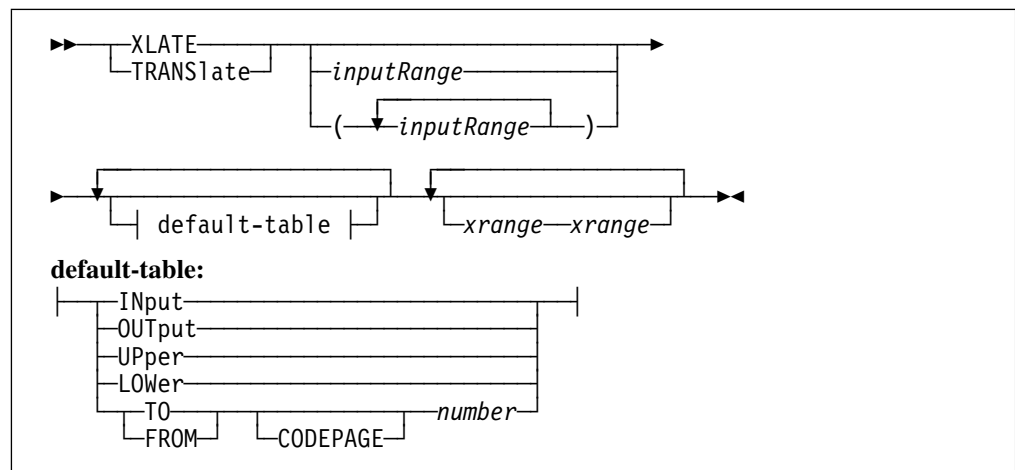
fanout then passes the input record to its secondary output stream, which is connected to the *xedit* stage. This causes the contents of the current line to be replaced by the contents of the record, in effect adding the record to the file. XEDIT then advances the current line to the next line. To compensate for this, the record is then turned into an XEDIT command to back the current line up to the one just inserted.

One final point: This example assumes that the file has variable record format. If the file is fixed record format, the record that is passed to XEDIT must be as long as the record length set for the file. Use *chop* and *pad* to coerce the record into the correct format.

4. The RANGE and SCOPE settings control which records are read from the file or replaced. Only lines with a selection level in the range between the limits set by the DISPLAY XEDIT subcommand are made available or replaced by *xedit* when SCOPE is DISPLAY. When a range is set by the RANGE XEDIT subcommand, only lines within that range are available or replaced. It appears that the selection level for lines added or replaced is set to the lowest value in the display range.
5. Set the file mode to S to be sure that a work file is not stored on disk accidentally.
6. Multiple *xedit* stages processing the same file are not recommended. XEDIT advances the line pointer after a line is read or written; it is difficult to predict the order in general.
7. XEDIT does not document which settings have any effect for the underlying interface.

xlate—Transliterate Contents of Records

xlate transliterates the contents of records in accordance with a translate table.



Type: Filter.

Syntax Description: The parameter list to *xlate* contains three sections:

- One or more input ranges, which specify the part of the record that is to be transliterated.
- One or more keywords that specify default translate tables. A composite table is constructed that maps directly from the input of the first table to the output of the last one.
- One or more hexadecimal ranges, which specify modifications to the default mapping. This modification is applied to the composite table; the modified translations replace the ones in the default table.

All three parts are optional.

An input range or a list of input ranges in parentheses is optional as the first argument.

One or more translate tables are optional after the input ranges. When more than one translate table is specified, the resulting table is the cumulative effect of the tables specified, in the order specified. The upper case table is used if there are no arguments or only input ranges and the secondary input stream is not defined. The neutral table is used

if there are additional arguments and no keyword is recognised for the default table. You may use an upper to lower table or one of the three CMS translate tables (upper case, SET INPUT, and SET OUTPUT). The tables for INPUT and OUTPUT default to the neutral one when no such SET is in effect.

When LOWER is used, the lower case translation table is constructed as the inverse of the CMS upper case translation table. If the upper case translate table translates two or more characters to a particular upper case one, the character with the lower hex value is used in constructing the upper to lower table.

Tables translating between codepage 500 and one of the national use codepages are provided with the keywords TO and FROM. Figure 401 shows the supported codepages. The first column contains the codepage number; the second column contains the base type (EBCDIC or ASCII); and the last column contains the country name (where it is known).

37	EBCDIC	United States, Canada, Netherlands, Portugal, Brazil, Australia, New Zealand.
37-2	EBCDIC	The real 3279 APL codepage, as used by C/370. This codepage is very close to 1047, except that the caret and the not sign are inverted. Codepage 37-2 is not recognised by IBM, even though SHARE have repeatedly pointed out its <i>de facto</i> existence.
273	EBCDIC	Austria, Germany.
274	EBCDIC	Belgium.
275	EBCDIC	Brazil.
277	EBCDIC	Denmark, Norway.
278	EBCDIC	Finland, Sweden.
279	EBCDIC	
280	EBCDIC	Italy.
281	EBCDIC	Japan.
282	EBCDIC	Portugal.
283	EBCDIC	
284	EBCDIC	Spain, Latin America.
285	EBCDIC	United Kingdom.
297	EBCDIC	France.
437	ASCII	PC Display: United States, Switzerland, Austria, Germany, France, Italy, United Kingdom.
500	EBCDIC	Belgium, Canada, Switzerland, International Latin-1. International number 5. This is the base codepage for <i>xlate</i> .
819	ASCII	ISO 8859 Latin Character Set 1 (Western Europe).
850	ASCII	PC Data-190: Latin Alphabet Number 1; Latin-1 countries.
863	ASCII	PC Display: Canada.

865	ASCII	PC Display: Denmark, Norway.
871	EBCDIC	Iceland.
1047	EBCDIC	C/370 variant of codepage 37, which takes into account the encoding of, for example, brackets that was inherited from the 3270 display system.

You can modify the translation further with translation elements, which map a “from” character or range into a “to” character or range. Each element of a from/to pair may be specified as a character, a two-character hex code, or a range of characters (*xrange*). Modifications to the starting translate table are made in the order they appear in the argument list. A character can be specified more than once; the last modification is the one that is used.

If a range is specified for the “from” part of a translation element and the “to” range is shorter than the “from” range, the last part of the “from” range is translated to the last (or only) character of the “to” range. That is, the last character is “sticky”. For example, 00-02 0-1 causes X'00' to be translated to 0; both X'01' and X'02' are translated to 1 (=X'F1').

Operation: If the secondary input stream is connected, a record is read from it before the primary stream is processed. The first 256 characters of this first record are used as the initial translate table, which is then modified as described above.

For each record on the primary input stream, *xlate* builds an output record with the same length as the input record.

The contents of the input record are copied into a buffer. Input ranges are then processed in the order specified in the first argument; the contents of a column are replaced by the corresponding value from the translate table. Depending on the contents of the table, multiple translates may change a character to a different character than the original translation. A column outside all ranges is left unchanged.

Streams Used: If the secondary input stream is defined, one record is read and consumed from it. The secondary input stream is severed before the primary input stream is processed. The secondary output stream must not be connected.

Record Delay: *xlate* strictly does not delay the record.

Commit Level: *xlate* starts on commit level -2. It verifies that the secondary output stream is not connected and then commits to level 0.

Premature Termination: *xlate* terminates when it discovers that its output stream is not connected. The corresponding input record is not consumed.

Examples: To remove punctuation and other special characters except single quotes:

```
... | xlate *-* 40-7f blank ' ' | ...
```

Modifications replace the default translate table; they are not performed in addition to this table:

```

pipe literal abcABC | xlate upper a z | console
▶zBCABC
▶Ready;
pipe literal abcABC | xlate e2a a z | console
▶zÃÄ ää
▶Ready;

```

Zoned decimal data are just like normal numbers, except that the sign is encoded in the leftmost bits of the rightmost digit. Positive numbers are indicated by a “digit” that is “A” through “I” for 1 through 9, respectively. Negative numbers are represented by “J” through “R”. (And presumably one should interpret 1 through 9 themselves as unsigned.) A zero digit is represented by the national use characters X'CO' and X'D0', respectively.

ZONE2DEC REXX, which is shown in Figure 402, is a sample filter to convert zoned decimal data in columns 1 through 5 to humanly readable format:

<i>Figure 402. ZONE2DEC REXX Converts Zoned Decimal to Normal Decimal</i>	
<pre> /* Convert columns 1-5 to normal decimal. */ Signal on novalue 'callpipe (name ZONE2DEC.REXX:4)', ' *:', ' spec 5 1 1-* 2', ' xlate 1 00-ff blank', 'CO-I +', 'D0-R -', ' xlate 6', 'CO-I 0-9', 'D0-R 0-9', ' *:' exit RC </pre>	<pre> /* Sign to 1 1-5 to 2-6 */ /* Handle sign; default unsigned */ /* These get plus ... */ /* and these are negative */ /* Now do the loworder digit */ /* Remove positive sign */ /* and remove negative sign */ </pre>
<pre> pipe literal 12345 1234E 1234N 0000{ split zone2dec console ▶ 12345 ▶+12345 ▶-12345 ▶+00000 ▶Ready; </pre>	

Notes:

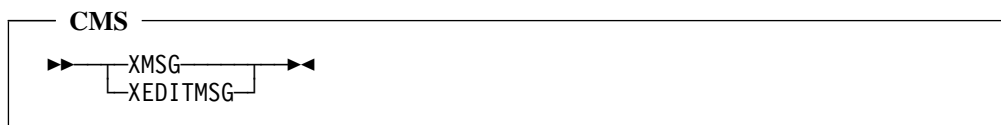
1. For compatibility with the past, *xlate* A2E is equivalent to *xlate* from 819. *xlate* E2A is equivalent to *xlate* to 819. These represent mappings between codepage 500, the international base codepage, and codepage 819, ISO 8859 Latin Character Set 1 (Western Europe).
2. Use a placeholder range **** when the complete record is to be translated with modification to the neutral table; this is *de rigueur* when the first “from” range is a valid *range* (for instance 40), but it is a good habit to use the placeholder even when the first “from” range cannot be taken for a *range*.
3. Modifications to the default table specify the direct input to output mapping; they are not performed after the default mapping. Beware, in particular, when the default table translates between EBCDIC and ASCII.
4. Use a cascade of *xlate* stages to perform stepwise translation. You can also compute a composite translate table by sending the neutral table (from *xrange*) through the cascade of *xlate* stages. Then provide this table on the secondary input stream to a

single *xlate* stage. This may give a marginal improvement of performance for large files.

5. The location of input ranges is computed based on the original input record. For example, translating blanks to a non-blank character does not change the position of a particular word.

xmsg—Issue XEDIT Messages

xmsg displays input lines as messages in the current XEDIT session.



Type: Device driver.

Operation: Lines are prefixed `msg` and then directed to XEDIT as commands, so that the text will be displayed on the XEDIT screen (as determined by the `SET MSGLINE XEDIT` subcommand).

Streams Used: *xmsg* passes the input to the output.

Record Delay: *xmsg* strictly does not delay the record.

See Also: *subcom* and *xedit*.

Examples: This XEDIT macro runs the arguments as a pipeline, writing its output as XEDIT messages:

```
/* PIPE XEDIT */
address command 'PIPE' arg(1) '| xmsg'
```

Notes:

1. The number of message lines that can be displayed on the XEDIT screen is controlled by the XEDIT command “`set msgline n m overlay`”. XEDIT reverts to line mode output when more messages are queued than can be displayed on the screen.
2. XEDIT may not display the message immediately. Use the `REFRESH XEDIT` subcommand to show the queued messages. You can turn the output from *xmsg* into refresh requests and issue these with *subcom* XEDIT, but more craftiness is required to display more than one message at a time and be sure the messages stay on the screen long enough to be read. Experiment with this kind of pipeline:

```
/* Issue output as messages and wait */
'PIPE ...
  | elastic',           /* Buffer messages while we wait */
  | xmsg',             /* Issue messages */
  | spec read read read read /command refresh/ 1', /* See below */
  | subcom xedit',     /* Issue refresh command */
  | spec /+5/ 1',     /* Make it a delay */
  | delay',           /* Wait five seconds */
  | hole'             /* Done; must stay connected */
```

The *elastic* stage decouples the pipeline segment shown from the rest of the pipeline, which can proceed even though *xmsg* may not be ready to read more messages. The messages are issued with *xmsg*. *spec* is used to drop four records (the four `READ`

keywords) and transform every fifth record into a refresh command which the *subcom* stage issues to XEDIT. The *subcom* stage also passes the record to the next stage, *spec*, which transforms it to the string “+5”. When the following *delay* stage reads this record, it causes the pipeline to wait for five seconds. The *xmsg* stage also waits for five seconds before reading more messages because none of the stages in this partial pipeline delay the record.

xpndhi—Expand Highlighting to Space between Words

xpndhi manipulates the contents of descriptor records produced by *overstr* such that a run of blanks between highlighted characters is also made highlighted.



Type: Arcane filter.

Syntax Description: A number is optional. It specifies the maximum number of contiguous blank characters to be turned into highlighted blank characters. The default is one.

Operation: Records that do not have X'00' in column one are copied unchanged to the output.

Descriptor records (having X'00' in the first position) in a data stream in the format of the output from *overstr* are changed to make spaces between highlighted words highlighted as well.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *xpndhi* does not delay the record.

Premature Termination: *xpndhi* terminates when it discovers that its output stream is not connected.

Notes:

1. *xpndhi* is intended to be used between *overstr* and *buildscr* when the resultant data are to be displayed in reverse video on a 3270 terminal.

xrange—Write a Range of Characters

xrange writes a single output record containing characters from one hexadecimal value to another one.



Type: Device driver.

Placement: *xrange* must be a first stage.

zone

Syntax Description: If the argument string is a single word, it is scanned for a range of characters. Otherwise it must be two words, which specify the beginning character followed by the ending character.

See Also: *literal*.

Examples: To write the first nine letters:

```
pipe xrange a-i | console
▶abcdefghi
▶Ready;
pipe xrange a.9 | console
▶abcdefghi
▶Ready;
pipe xrange a i | console
▶abcdefghi
▶Ready;
```

Note that the output record contains all values in the range:

```
pipe xrange i j | console
▶i«»%~ijff°j
▶Ready;
```

You can use two hexadecimal digits:

```
pipe xrange f0 f9 | console
▶0123456789
▶Ready;
```

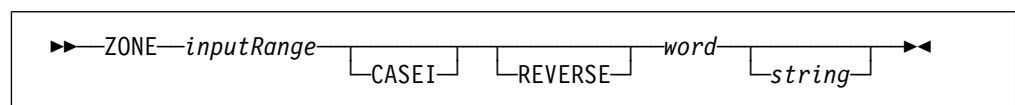
The range wraps from X'FF' to X'00':

```
pipe xrange fe 01 | spec 1-* c2x | console
▶FEFF0001
▶Ready;
pipe xrange 9 0 | count bytes | console
▶248
▶Ready;
```

zone—Run Selection Stage on Subset of Input Record

The argument to *zone* is a stage to run. *zone* invokes the specified stage. It then passes a specified range of the input records to this stage. When the stage writes a record to its primary output stream, the corresponding original input record is written to the primary output stream from *zone*; likewise, the original input record is written to the secondary output stream (if it is defined), when the stage writes to its secondary output stream.

The argument is assumed to be a selection stage; that is, it should specify a program that reads only from its primary input stream and passes these records unmodified to its primary output stream or its secondary output stream without delaying them.



Type: Control.

Syntax Description: An input range and a word (the name of the program to run) is required; further arguments are optional as far as *zone* is concerned, but the specified program may require arguments.

Operation: *zone* constructs a subroutine pipeline to perform the required transformation on input records. If CASEI is specified, the specified part of the input record is translated to upper case before it is passed to the specified stage (see *casei*). If REVERSE is specified, the contents of the zone are reversed before being passed to the specified stage.

Streams Used: Records are read from the primary input stream; no other input stream may be connected.

Record Delay: *zone* does not add delay.

Commit Level: *zone* starts on commit level -2. It does not perform an explicit commit; the specified program must do so.

See Also: *casei*, *predselect*, and *spec*.

Examples: To select records between those that end in “on” and “off”:

```
pipe ... | zone -3;-1 reverse between /no/ /ffo/ | ...
```

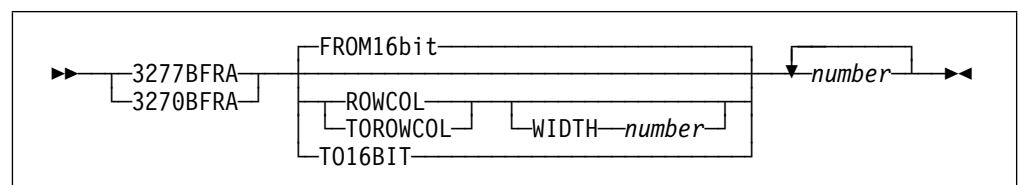
Notes:

1. Use *asmfind* or *asmnfind* as the argument to *zone* only if you understand the implications of shifting the continuation column in the input records that the selection stage reads.
2. The argument string to *zone* is passed through the pipeline specification parser only once (when the scanner processes the *zone* stage), unlike the argument strings for *append* and *preface*.
3. End-of-file is propagated from the streams of *zone* to the corresponding stream of the specified selection stage.

Return Codes: If *zone* finds no errors, the return code is the one received from the selection stage.

3277bfra—Convert a 3270 Buffer Address Between Representations

3277bfra converts a halfword buffer address between two of three possible representations.



Type: Arcane filter.

Syntax Description:

FROM16BIT	A binary 12-bit buffer address is encoded in two printable characters. If the four leftmost bits are not zero, the buffer address remains unchanged.
ROWCOL	The two bytes are interpreted as a 1-byte row and column, respectively. The converted result is a 12-bit encoded buffer address when it is less than 4096; it is a 16-bit binary buffer address otherwise.
TOROWCOL	The buffer address is first converted to a binary address as done for TO16BIT. The binary buffer address is then divided by the screen width to obtain the row and column; these are each truncated to 8 bits and combined with the row in the leftmost 8 bits.
TO16BIT	Convert a 12-bit buffer address from encoded form to binary. If the two leftmost bits of both bytes are nonzero, a 12-bit binary address is extracted from the rightmost six bits of the two bytes. If either of the two bytes contain zeros in the two leftmost bits, the buffer address remains unchanged.
WIDTH	Specify the screen width. The default width is 80.

The final numbers specify the columns where the buffer addresses begin. Up to ten numbers may be specified.

Record Delay: *3277bfra* strictly does not delay the record.

Premature Termination: *3277bfra* terminates when it discovers that its output stream is not connected.

Examples: To generate a set buffer address order for each line on a screen:

```
/* Build screen */
'PIPE (name 3277BFRA)',
  '|stem data. ',
  '|spec x11 1 number from 0 by 80 d2c 2.2 right 1-* next',
  '|3277bfra 2',
  '|join *',
  '|var buffer'
```

To reverse this process on an inbound 3270 data stream:

```
/* Deblock inbound data (read modified) */
parse var inbound AID+1 cursor+2 fields
'PIPE (name 3277BFRA)',
  '|var fields',
  '|split minimum 3 before 11',
  '|3270bfra 2 to16bit',
  '|...'
```

Notes:

1. For compatibility with previous releases, the operands can also be specified as a single number followed by the keyword TO16BIT.

3277enc—Write the 3277 6-bit Encoding Vector

3277enc writes a single record containing 64 characters.



Type: Arcane device driver.

Placement: *3277enc* must be a first stage.

Operation: A 64-byte record is written. The first byte contains the encoding for X'00'; the second byte contains the encoding for X'01'; and so on.

Examples: To encode an attribute byte:

```
/* Get encoding vector */
'PIPE 3277enc | var encoder'
attr=translate(attr, encoder, xrange("00"x, "3f"x))
```

The encoding string can also be fed to *xlate*'s secondary input stream to be used to translate a stream of records that contain an attribute value followed by data. Assume that the input record contains three fields: the first two columns contain the buffer address (16-bit addressing); the third position contains the attribute byte; and the remainder of the record contains the data to be put into a field having the specified attribute at the specified address.

The buffer address and the attribute byte are processed according to their natures. The data part of the record is then translated to ensure it cannot interfere with the device orders. Finally, the 3270 device orders are inserted into the data stream.

```
'callpipe (end ? name 3277ENC)',
'|*:', /* Input file */
'|3277bfra 1', /* Make buffer address "printable" */
'|x: xlate 3', /* Make attribute "printable" */
'|xlate 4-* 01-3f blank ff blank', /* Rub out other controls */
'|spec x11 1 1.2 next x1d next 3-* next', /* Orders */
'|*:', /* Write to output */
'?3277enc', /* Get encoding vector */
'|x:' /* Pass it to XLATE. */
```

64decode—Decode MIME Base-64 Format

64decode decodes records that have been encoded according to RFC 1521.



Type: Filter.

Operation: *64decode* produces one output record for each input record.

Input Record Format: The encoded data are represented in EBCDIC. The records should be a multiple of four bytes in length, but this is not enforced.

64encode

Record Delay: *64decode* does not delay the record.

Premature Termination: *64decode* terminates when it discovers that its output stream is not connected.

64encode—Encode to MIME Base-64 Format

64encode encodes binary data according to RFC 1521.



▶▶64ENCODE▶▶

Type: Filter.

Input Record Format: A byte stream.

Output Record Format: 76-byte records, except possibly for the last.

Streams Used: Records are read from the primary input stream and written to the primary output stream. Null input records are discarded.

Record Delay: *64encode* does not delay the record.

Premature Termination: *64encode* terminates when it discovers that its output stream is not connected.

Chapter 24. *spec* Reference

This chapter contains the definitive syntax reference for *spec*. Chapter 16, “*spec* Tutorial” on page 166 contains a tutorial. The inventory of built-in programs contains an article on *spec*, which contains a simplified syntax description.

This chapter contains a description of the syntax of *spec* in the form of annotated syntax diagrams. Instead of presenting a single enormous syntax diagram followed by explanation, this chapter intermixes syntax diagrams and their explanation.

Overview

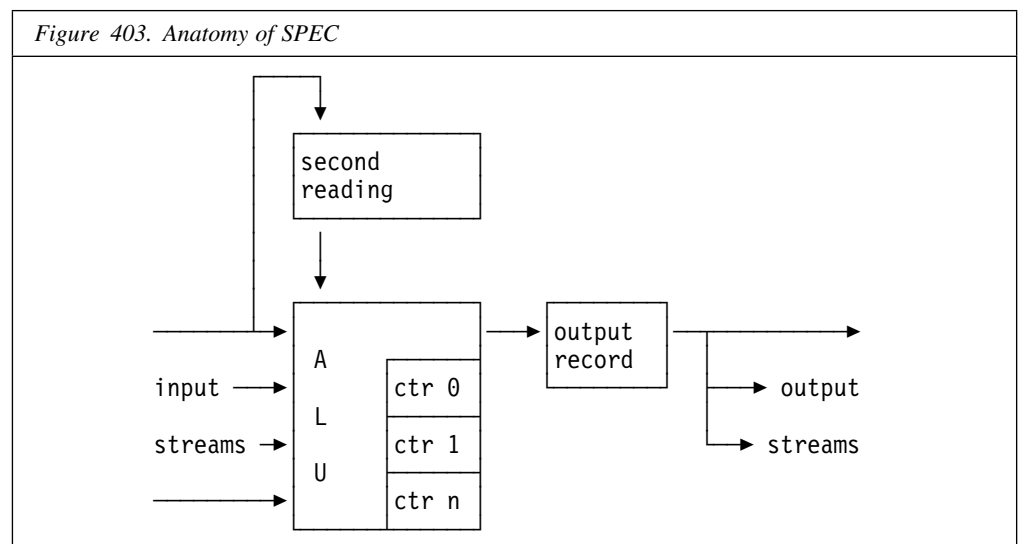
spec builds output records based on the contents of input records, constant fields, and data generated internally. *spec* supports any number of input and output streams.

The format of the output record is specified by a list of items, one for each field; thus the list is called a *specification list*.

You can think of the specification list as a program that is run for each input record. Each item in the list is a step in this program. Performing the action specified in an item is called *issuing* the specification item. Specification items that are not issued are *ignored*.

spec ignores case in keywords, but respects case in stream identifiers, field identifiers, and in literal strings.

spec can compute numeric expressions, compare fields and numbers, and issue items conditionally. It stores values in counters, which persist from record to record.



Though *spec* clearly does not require specialised hardware, it can be viewed as a software simulation of a hardware architecture. (Indeed, all programs can.) Figure 403 shows the parts of *spec*, that you, the programmer of this abstract machine, can reach.

Rest assured that no hardware knowledge is required to use *spec* effectively. But you will have a head start if you have past experience with the IBM 407 Accounting Machine, which influenced the design of *spec*.

If you have experience with RPG, most of the concepts will be familiar as well; this is no coincidence, because RPG, too, has its roots in accounting machines.

Concepts

The Cycle

To perform a *cycle*, *spec* issues the items in the specification list from left to right. At the beginning of a cycle, *spec* synchronises the input streams it uses; that is, it peeks at all input streams it uses before issuing any specification items. This ensures that a record is available on all streams at the beginning of the cycle. At the end of a cycle, *spec* consumes an input record from each of the streams it uses.

In addition to the streams, *spec* can store the previous record from the primary input stream into a buffer known as the *second reading station* and make this available as a pseudo input stream on the next cycle. The primary input stream is also known as the *first reading station*.

Streams

The primary input stream is selected at the beginning of the cycle unless a SELECT item precedes the first item that refers to an input range. With this proviso, *spec* uses all streams that are mentioned in SELECT items, even when no data field is referenced from the stream. BREAK, EOF, SELECT SECOND, and break() imply reading the primary input stream because they refer the second reading, which is derived from the primary input stream.

At the beginning of each cycle, *spec* sets the primary output stream as the one to receive the next record. This can be modified by the OUTSTREAM and NOWRITE items.

When the second reading is the only referenced stream, *spec* reads the first record from the primary input stream directly into the second reading before taking the first cycle, having the second record at the primary input stream.

When both the first and the second reading are used, *spec* performs an initial *runin cycle* with the first record at the primary input stream and a null record at the second reading. Any specification items that are subject to SELECT SECOND are ignored during the runin cycle. This prevents a spurious subtotal.

When the EOF item or the second reading is used, *spec* performs a final *runout cycle* after it comes to end-of-file on its input; the first reading contains a null record during this cycle (and the second reading contains the previous record on the primary input stream).

If an EOF specification item is present, it will be the first issued during the runout cycle; otherwise the normal sequence is issued. In either case, any specification items that are subject to SELECT FIRST are ignored during the runout cycle to prevent a spurious heading.

Field Identifiers, Control Breaks, Break Levels

A *field identifier* is associated with a specification item when an input range (which represents data from an input record) is prefixed by a letter and a colon. This allows for subsequent reference to the input range without repeating the actual specification. The field identifier is a single alphabetic character; case is respected.

The scope of a field identifier is from the specification item where it is defined to the next READ or READTO, or to the end of the specification list. Field identifiers cannot be defined in specification items that are issued conditionally; that is, after EOF or BREAK; or within an IF or WHILE group.

When a field identifier is referenced in a BREAK item or by the built-in function break(), a test is performed to determine whether a break is established for this level or not. This is determined at the time the identified item is issued, not at the time of the test. The break levels are ordered from “a” (lowest) through “z” and further from “A” through “Z” (higher). End-of-file represents an even higher break level.

When a specification item is issued and it has an identifier that is used to control breaks, the fields are compared as follows:

- If SELECT SECOND is in effect for the item, the contents of the equivalent field on the primary input stream are compared with the field specified in the item. That is, the first and second readings are compared.
- If some other select is in effect, the field is compared with the equivalent field in the second reading station. Normally, the primary input stream would be selected to allow consecutive records to be tested, but it is not an error to test some other input stream; maybe it contains the same data as the corresponding record on the primary input stream.

A break at some level also establishes the break at all lower levels, but the break must be established before it takes effect. That is, the specification item that defines the field that causes the break to be established must have been issued.

Structured Data

Fields in input and output records may be referenced as members of structures (see Chapter 6, “Processing Structured Data” on page 91).

A data type may be associated with a member of a structure, specified by a single character. The types recognised by *spec* are:

C	A character string.
D	Binary integer (big endian) in two’s complement notation. The input field may have any length.
F	System/360 hexadecimal floating point. The input field may have any length, but only the first sixteen bytes are used (corresponding to extended precision). If it is present, the eighth byte is ignored; it is the characteristic of the lower half.
P	System/360 packed decimal integer. A scale may optionally be associated with the member by specifying a signed number in parentheses. A positive scale specifies the number of decimal places; a negative one specifies the number of integer digits to drop on the right.
R	Byte-reversed (little endian) binary integer in two’s complement notation. The input field may have any length.
U	Unsigned binary integer. The input field may have any length.

A blank type, which means no type defined, as well as other letters are treated the same as character strings.

: Note that explicit conversion in a specification item disables a numeric member type
: including any scale; an output placement that specifies a member will contribute only the
: position and field length.

Counters

: Data, numbers as well as strings, are stored in *counters*. Counters are referenced by inte-
: gers that are zero or positive. There is no arbitrary limit to the number of counters; and
: you should not worry overly about which numbers you use. Do avoid, however, a large
: range of unused counter numbers.

: The counters are initialised to contain a null value (which converts to 0 or a null string, as
: appropriate for the context) when *spec* starts; after that they are set only as a side effect of
: evaluating expressions.

: A particular counter may hold a number at some time and be assigned a string at other
: times. That is, counters are not declared to contain a particular data type.

Number Representation

: Arithmetic is done in signed decimal floating point with thirty-one significant digits and an
: exponent range from -2G to 2G-1. (This is not the decimal floating point recently intro-
: duced in IBM's Power Systems and System z machines.) For division, the significance of
: the divisor is reduced to fifteen digits due to the System/390 hardware implementation.
: This may affect the precision of division, integer division, and remainder operations.

: In addition to the numeric value, an *exactness latch* is associated with the contents of a
: counter. This latch can be tested with the `exact()` built-in function; it is set in several
: ways:

- : • When a string is converted to load into a counter. The counter will be exact unless
: more than thirty-one significant digits are present and a nonzero digit is dropped.
- : • Assignment from a member that has D type is always exact.
- : • Assignment from a member that has F type is also exact, even though the input data
: may have lost exactness at an earlier time. For example, 1.3 cannot be represented as
: an exact hexadecimal floating point number.
- : • Assignment from a member that has P type first sets the exactness from the sign of the
: input field. Exactness is lost when significant digits are truncated as part of the
: assignment.

: The sign nibble A represents an exact positive number; B represents an exact negative
: one; D represents inexact negative; and the three remaining ones all represent inexact
: positive.

: Exactness is exposed on output to a member that has the packed type; truncation of
: significant digits in this assignment will make the output field inexact even when the
: source number is exact.

When an expression is evaluated to control the issuing of specification items, a zero result
represents the Boolean value `false`; other values represent the Boolean value `true`.

Expressions

spec supports numeric and string expressions.

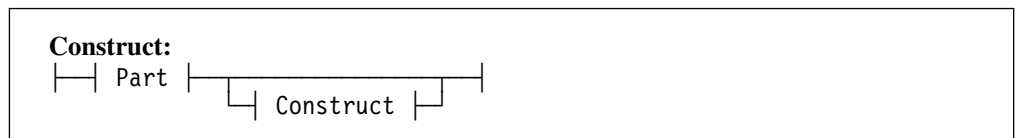
spec computes expressions using data from input records, constants, and the contents of counters. The result can be stored in a counter or formatted for printing, or both. *spec* can compare the contents of counters, the contents of an identified input field, and literals in any combination. Comparison is numeric or by character, as determined by the operator.

The result of an expression can alter the flow through specification items to issue items selectively (if/then/else) or repeatedly (while).

Syntax Recursion

While it may be a strange programming language, the arguments to *spec* support many concepts that are normally associated with programming languages. Supporting general constructs, such as nested conditional statements or expressions whose terms can be expressions in their own right, invariably leads to recursive syntax diagrams.

But let them not intimidate you; the infinities are easily resolved. Consider this diagram:

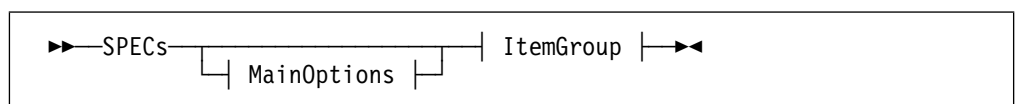


Such a syntax diagram should be read in this way: A construct must contain one part, because the part is on the main track. After the part you have a choice; you can end the construct right there by taking the straight exit line; or you can make a recursion by taking one more construct, which would lead you to one more part, and so on for as long as you wish.

Syntax Description

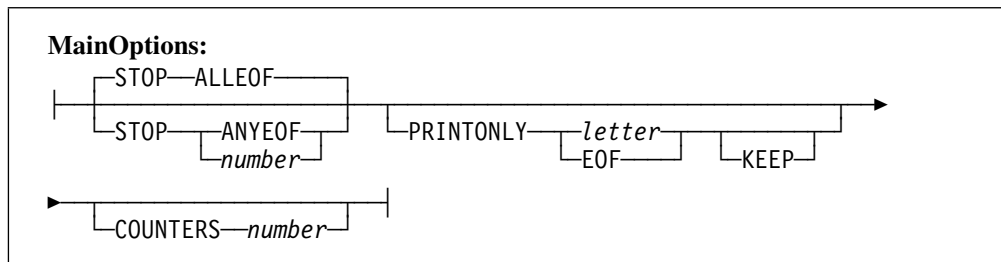
Syntax Overview

The arguments to *spec* contain options followed by a group of specification items. The options must be first in the argument string.



Main Options

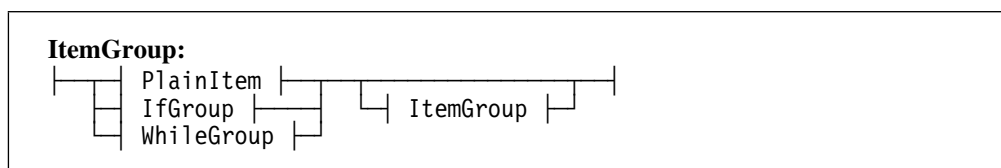
Keywords that control the general behaviour of *spec* are specified at the beginning of the argument string.



- STOP Specify when *spec* should terminate. ALLEOF, the default, specifies that *spec* should continue as long as at least one input stream is connected. ANYEOF specifies that *spec* should stop as soon as it determines that an input stream is no longer connected. A number specifies the number of unconnected streams that will cause *spec* to terminate. The number 1 is equivalent to ANYEOF. STOP must be specified first.
- PRINTONLY Output records will be suppressed unless the specified break level has been established. The break level can be a letter (case is respected) or it can be the keyword EOF, which specifies that records are written only after the input reaches end-of-file or the condition specified with STOP is satisfied.
- KEEP Do not reset the output buffer when write is suppressed due to the break level. This allows the contents of several input records to be made present in a single output record.
- COUNTERS Specify the largest counter number referenced as an array member. *spec* allocates at least counters 0 through the number specified.

Item Group

An item group is a list of specification items that are issued sequentially. Each item in a group can be a plain item, an IF group, which is a group from which some items are issued and others are suppressed, or a WHILE group, which is a group that is issued repeatedly while an expression evaluates to true.

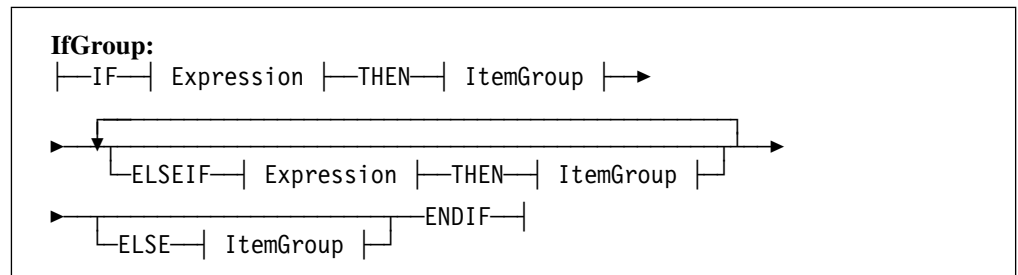


If Groups and While Groups are known collectively as conditional groups. They can nest one inside another in any combination to a depth of sixteen. There is no ambiguity of, for example, ELSE because each nested group is terminated with a distinct keyword (ENDIF or DONE).

Conditional groups cannot contain input sources that have identifiers, as it is indeterminate, in general, whether the field has been defined or not.

If Group

A group of specification items from IF to the matching ENDIF is called an IF group. Depending on the result of evaluating an expression, some of the specification items are issued and others are ignored. Because an IF group can contain several tests, it is similar to the REXX Select instruction.



The IF group contains four parts:

- The opening keyword and the first test to be performed:
 - The keyword IF.
 - An expression to be evaluated.
 - The keyword THEN.
 - An item group, which is issued if the result of the expression is true (that is, nonzero). Control then transfers to the end of the IF group, ignoring any remaining items. If the expression evaluates to false, the item group is ignored.
- Additional tests to perform if the initial test evaluated to false. The tests are evaluated in the order they appear until a nonzero result is obtained. This sequence is optional; it can occur zero or more times.
 - The keyword ELSEIF. (ELSIF is also recognised.)
 - An expression to be evaluated.
 - The keyword THEN.
 - An item group, which is issued if the result of the expression is true (that is, nonzero). Control then transfers to the end of the IF group, ignoring any remaining items. If the expression evaluates to false, the item group is ignored.
- The default case. This sequence is optional; it can occur at most once.
 - The keyword ELSE. (OTHERWISE is also recognised.)
 - An item group.
- The ending sequence.
 - The keyword ENDIF. (FI and IFEND are also recognised.)

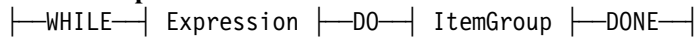
: ELSE IF is also recognised, but this starts a new IF group that is nested within the ELSE
 : group. The nested IF group could be followed by other specification items as part of the
 : ELSE group.

: While Group

: A group of specification items from WHILE to the matching DONE is called a WHILE group.
 : Depending on the result of evaluating an expression, the specification items in the group
 : are either skipped or issued repeatedly until the expression evaluates to false.

: Ensure that the loop always terminates. If it does not, PIPMOD STOP ACTIVE will terminate
 : spec.

WhileGroup:



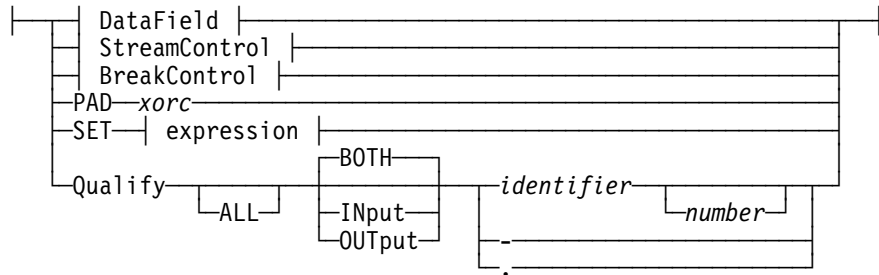
The WHILE group contains two parts:

- The opening keyword and the test to be performed:
 - The keyword WHILE.
 - An expression to be evaluated.
 - The keyword DO.
- An item group, which is issued if the result of the expression is true (that is, nonzero). The group is terminated by the DONE keyword.

Plain Item

A plain item can describe a field in an output record; it can control input and output streams; it can react to control breaks; it can specify a pad character; it can specify an expression to evaluate for its side effects; and it can establish or disable a qualifier for use in specifying members of structures.

PlainItem:



PAD Specify the character to insert in the output record between output fields. The default pad character is the blank.

SET Specify an expression to be evaluated for its side effects. (That is, to set variables.) The result is discarded.

QUALIFY Specify the qualifier for all streams or the currently selected stream of the specified type, or both input and output. The number specifies the beginning column for the structure; column 1 is the default. You must specify a number when the next item is an input source that also is a single column. A period or a hyphen disables any active qualifier.

Specify **BOTH** explicitly when the identifier scans as one of the input/output keywords.

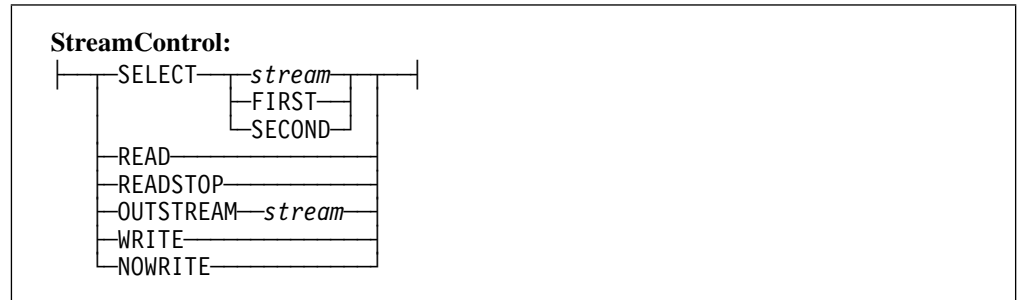
The **QUALIFY** item can be specified anywhere in the item list; in particular, a member need not follow, as it must when used in an *inputRange*.

Notes:

1. Using a qualifier that applies to a particular input stream is definitely useful when several members are referenced in a number of specification items interspersed with **SELECT** items.

Stream Control

The stream control items read or write records during the cycle; they select the source of data for subsequent input fields; and they select the stream to which subsequent writes will be directed.



SELECT	Select the source for input data in subsequent field items. Specify the stream identifier for the input stream you wish to process.
.	Or specify one of the keywords FIRST and SECOND to use the record in that particular reading station. SELECT FIRST is a convenience for SELECT 0.
.	SELECT 0 is implied at the beginning of the cycle unless some other SELECT item precedes the first data field.
READ	Consume the record on the currently selected input and peek at the next input record. For READSTOP, terminate the cycle if the stream is at end-of-file. Otherwise a null record is assumed if the stream is at end-of-file. This record is used as the source of input fields in subsequent field items. The second reading station is unaffected. All field identifiers become undefined. The break level is reset to no break established.
READSTOP	
:	READ and READSTOP for the primary input stream discards the current record; it will not be available from the second reading.
:	READ and READSTOP are not valid after SELECT SECOND as there is no stream to read.
OUTSTREAM	Specify the stream to receive output records. Subsequent WRITE items will write to the specified stream. The last OUTSTREAM item issued in a cycle specifies where to write the record at the end of the cycle.
WRITE	OUTSTREAM 0 is implied at the beginning of a cycle.
WRITE	Write the output record built so far to the stream specified in the last OUTSTREAM item issued. Reset the output record to be null.
:	Suppress the write at the end of the cycle. NOPRINT is a synonym for NOWRITE
NOWRITE	

Break Control

Specification items after a break control are issued only if a break level has been established at least to the level specified.

Break items are not allowed in IF groups. The break items are a convenience; they can also be formulated with IF.

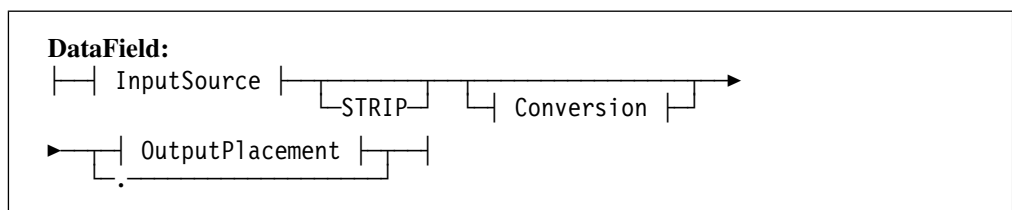


BREAK <i>letter</i>	The subsequent specification items are issued only if a break has been established at the specified level or higher. Otherwise all items up to the next break control are ignored. Further break items are allowed.
BREAK FIRST	The subsequent specification items are issued only on the runin cycle. Further break items are allowed.
BREAK EOF	The subsequent specification items are issued only on the runout cycle. Further break items are allowed.
EOF	The remainder of the specification list is issued only on the last cycle performed by <i>spec</i> ; it is ignored on other cycles. No further break items are allowed in the specification list. If no runout cycle is scheduled; that is, the second reading is not used, a final cycle is taken issuing only the items that follow EOF.

Data Field

A data field inserts a field in the output record. You must specify an input source and an output placement; you may also specify conversion of the contents of the field from one representation to another one; and you may specify that the input field is stripped of blanks before it is converted (or before it is placed in the output, if you specify no conversion).

A data field is suppressed if it refers to an input range that is not present in the record and the output placement does not specify an explicit length for the output field; the item is then ignored.

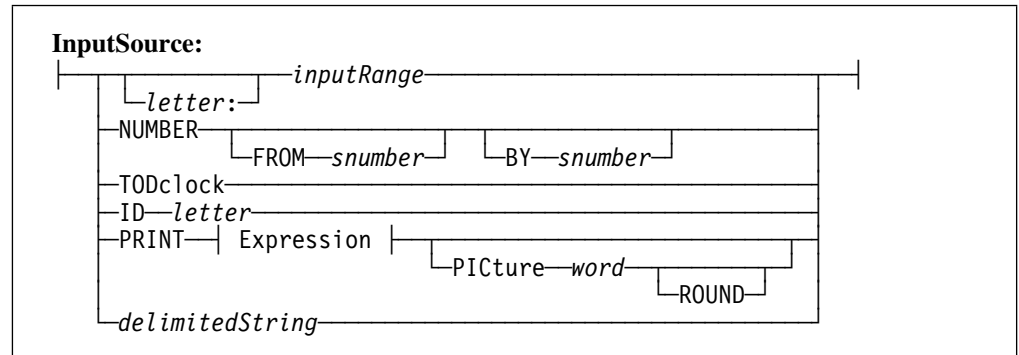


STRIP	Strip the input field of leading and trailing blanks for subsequent use within this specification item. Other specification items that reference this specification item through a field identifier use the original contents of the input range.
.	(A period.) Do not insert the field into the output record. The period makes sense only if the field originates in an input record and a field identifier is specified; specifying a period with other items has the same effect as omitting them, except that any side effects of evaluating a PRINT expression will still occur.

Input Source

The input source specifies where data come from. Data can originate in an input record; they can be the result of evaluating an expression; they can be a literal constant; or they can be generated internally in *spec*.

There is a length associated with an input field.



inputRange Data originate in an input record or specify a manifest constant.

Specify a substring of an input record to reference input data. The record used is the one selected by the last SELECT issued. A substring that is completely outside the record is considered null. (It contains no data.)

You can prefix the input range with a letter and a colon unless the item is in a conditional group. This identifies the field for use in subsequent expressions or ID items.

The implied field length is the length of the input field.

A manifest constant is syntactically a subset of an *inputRange*, but the structure defines the member as a constant rather than as a field. A manifest constant is not valid in conjunction with a field identifier. A manifest constant refers to four bytes binary (two's complement notation), which are of type D. The default placement is right aligned. Thus, vmcparm.vmcpsend resolves to a field that contains X'00000002'.

spec Reference

!	NUMBER	<p>Data originate in <i>spec</i>. When no additional keywords are specified, and NUMBER is not issued conditionally, the number generated is the number of the current cycle. That is, READ or READSTOP items do not increase the number generated.</p> <p>Two subkeywords are optional. FROM specifies the record number to use for the first record; the default is 1. BY specifies the increment; the default is 1. Numbers can be negative.</p> <p>The number is incremented each time the specification item is issued, and only then. Thus, when NUMBER is issued conditionally or in a break, it may not reflect the number of records processed by <i>spec</i>; it may be higher when in a WHILE group. Specifically, NUMBER that is issued on end-of-file, will not have been incremented previously. Use the number() built-in function to obtain the number of the current record.</p> <p>The record number is computed in decimal arithmetic with up to fifteen significant digits. Overflow is ignored; the carry out of the leftmost digit is lost; the sign remains unchanged. The number is aligned to the right with a drifting minus sign.</p> <p>The implied field length is ten characters.</p> <p>Each NUMBER item maintains a separate record number.</p>
:	TODCLOCK	<p>Data originate in <i>spec</i>. Eight bytes are provided containing the value of the time-of-day clock at the beginning of the cycle. The field contains a 64-bit binary counter; the thirty-first bit is incremented slightly less often than once a second. Refer to z/Architecture Principles of Operation, SA22-7832, for a description of the time-of-day clock.</p> <p>The implied field length is eight bytes.</p>
:	ID	<p>Refer back to the contents, before stripping, of a previously defined input range.</p> <p>The implied field length is the length of the input range.</p>
:	PRINT	<p>Compute an expression and format the result. You may specify PICTURE to supply the picture under which the result is presented. Refer to “Pictures” on page 747 for the syntax of a picture. The default picture is eleven digits with a drifting minus sign. When a picture is specified, the expression must evaluate to a numeric result; when the picture is omitted, a string expression is also acceptable. When a picture is used, the value is truncated unless ROUND is specified.</p> <p>The implied field length is the length of the picture (ignoring a V, if one is specified).</p> <p><i>delimitedString</i>The delimited string represents a constant in all cycles. The constant can be specified as a string between delimiter characters, as a hexadecimal literal, or as a binary literal.</p> <p>The implied field length is the length of the literal.</p>

Notes:

1. NUMBER is a convenience; the same result can be obtained by incrementing a counter.
2. No provision exists to store the complete time-of-day clock.

Conversions

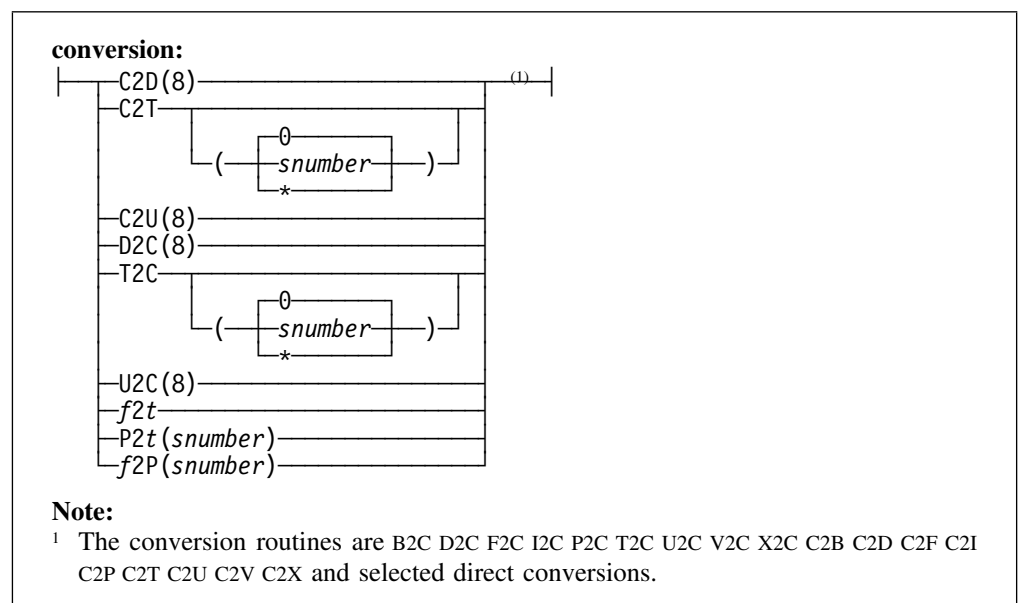
The conversions are modelled on the REXX built-in functions to convert between binary and other formats.

The conversion routine names are of the form *x2y*, where *x* represents the data format before conversion and *y* represents the format desired for the result.

The REXX function names may be confusing; if so, the confusion is carried over into *spec*. The confusion stems from the fact that the format *C* (for character) is not usually printable characters, rather it is the internal form of the data, as represented inside the computer.

Thus, *d2c* converts from printable decimal to four bytes of binary data. Possibly, you would have expected it to be the other way round, but there you are; REXX works the same way.

Note: Explicit conversion disables any numeric type specified for a member, both in the input source and in the output placement. In general, explicit conversion is incompatible with structured data.



These routines convert from internal representation (binary) to a format that can be printed or displayed on a terminal:

C2B Convert bytes to bit string (unpack bytes to bit strings). For each character in the input field, the result has eight bytes containing the character 0 or 1 (X'F0' or X'F1').

C2D Convert a binary integer using two's complement notation for negative numbers to a character string. C2D operates on 32-bit integers; C2D(8) operates on 64-bit integers. This is the format used for fixed point integers on IBM System/360 and its descendants. Numbers must be within the 32-bit (64-bit) precision of the fixed point instruction set; input fields longer than 4 (8) characters are allowed only if the leading characters represent a sign extension that can be stripped down to 4 (8) bytes. For C2D, the conversion result is 11 characters, aligned to the right, padded with blanks on the left. (It has no leading zeros.) C2D(8) produces a field that is large enough to contain the number, but, unlike C2D, no longer. That is, the number aligned to the left; be

sure to specify an output range with right alignment in tabular reports. Negative numbers have a leading minus; other numbers are unsigned.

- C2F Convert a doubleword of IBM System/360 long floating point representation (hexadecimal floating point) to scientific notation. The input field must be between two and eight bytes long. A field that is shorter than eight bytes is padded on the right with binary zeros. Exact zero (with either sign) is converted to the single character, 0. The conversion result for other numbers is a 22-character string containing the number in scientific notation. It has a leading sign, one significant digit before the decimal point, and a 15-digit fraction. Numbers with an absolute value from 1 to 10 have four trailing blanks. Numbers numerically 10 and larger are represented with a positive exponent; numbers numerically less than 1 are shown with a negative exponent.
- C2I Convert from z/OS Julian date format to ISO (or “sorted”) timestamp format. The input field may contain between three and seven characters. Each halfbyte contains a decimal digit, except for a sign (which must be X'F') in the rightmost four bits of the third or fourth byte. When the sign is in the third byte, the year is in the century beginning with the year 1900. When the sign is in the fourth byte, the first byte contains the number of centuries beyond 1900. Up to three bytes containing a timestamp are allowed after the sign. The output field contains at least eight characters for the date, expressed in ISO format: `yyyymmdd`. Additional characters are appended when the input field contains a timestamp.
- C2T Convert eight bytes binary time-of-day clock value to an ISO timestamp. A time zone offset in seconds can be specified as a signed number in parentheses; specify a positive number east of Greenwich. The default is 0. Specify * to use the time zone offset that CP stores on diagnose 0.
- C2P Convert a packed decimal number to a printable form. The input field must contain a valid IBM System/360 packed decimal number (but it is not restricted in length). A scale factor in parentheses may be appended to the name of the conversion routine. When no scaling is specified, the output field contains a sign (plus or minus) and an integer number. When scaling is specified, a positive scaling indicates the number of decimal places; a negative scaling represents additional orders of magnitude in the number. The output field contains a sign (plus or minus), the integer part of the number (if any), a decimal point, and the decimal fraction (if any).
- C2U Convert an unsigned binary integer. Except for sign, processing is identical to the corresponding variant of C2D.
- C2V Select substring. The input field is a varying length character string. It consists of a halfword that contains the length in binary (unsigned); this is followed by the characters of the string. The string length may be in the range 0 through 65535 inclusive. The conversion result begins with the third byte of the input field; it has as many characters from the input field as the halfword string length specifies. It is an error if the string length is larger than the length of the input field minus two.
- C2X Convert bytes to hexadecimal (unpack hex). The conversion result has two characters (0 through 9 and A through F) for each input character.

These routines convert a “readable” representation to the internal (binary) representation:

- B2C Pack bits. The input field must consist entirely of the characters 0 and 1; the length must be a multiple of 8. The result has one character for each 8 characters in the input field.

D2C	Convert from signed decimal to binary. The input field must contain a decimal integer that may be signed or unsigned. Leading and trailing blanks are allowed; blanks are allowed between the sign and the number. The result is 4 bytes (8 bytes for C2D(8)). with the number in binary using two's complement notation for negative numbers. The number must be within the 32-bit (64-bit) precision of IBM System/360 integer arithmetic.
F2C	Convert from decimal to hexadecimal floating point. The input field must contain the external representation of a floating point number which can be signed and can have an exponent. The result is 8 bytes containing the number in the format of a long floating point number in IBM System/360 hexadecimal floating point notation.
I2C	Convert from ISO timestamp to z/OS Julian date format. The length of the input field must be even, between six and fourteen. When the input field is six characters, three bytes output is generated. When the input field is eight characters or longer, the first four characters specify the year.
P2C	Pack a decimal number. The input field consists of an optional sign, an optional integer, and an optional decimal fraction. Blanks are allowed before and after the number, and between the sign and the number. A scale factor in parentheses may be appended to the name of the conversion routine. When no scale factor is present, the packed number contains all digits from the input field, right aligned; if present, a decimal point is ignored. When the scale factor is zero, the packed number contains only the integer part of the input field. When a negative scale factor is specified, that number of integer digits are truncated on the right (the fraction is ignored). When a positive scale factor is specified, the fraction is truncated or padded with zero on the right to this number of digits.
T2C	Convert an ISO timestamp to eight bytes time-of-day clock. A time zone offset in seconds can be specified as a signed number in parentheses; specify a positive number east of Greenwich. The default is 0. Specify * to use the time zone offset that CP stores on diagnose 0.
U2C	Convert from decimal to unsigned binary. Except for the lack of sign and the extended number range, processing is identical to the corresponding D2C variant.
V2C	Prefix field length. The result consists of an unsigned halfword (16 bits) with the length of the input field, followed by the contents of the input field. The longest acceptable input field is 65535 bytes.
X2C	Convert pairs of hexadecimal digits to single characters. The input field must contain an even number of <i>hex</i> characters. As for REXX hexadecimal constants, blanks are allowed at byte boundaries internally in the input field, but not at the beginning or the end. The result is a character string with one character for each two <i>hex</i> characters in the input field.

Some conversions are supported directly between printable formats, for example X2B. This table summarises the supported combinations. A blank indicates that the combination is not supported.

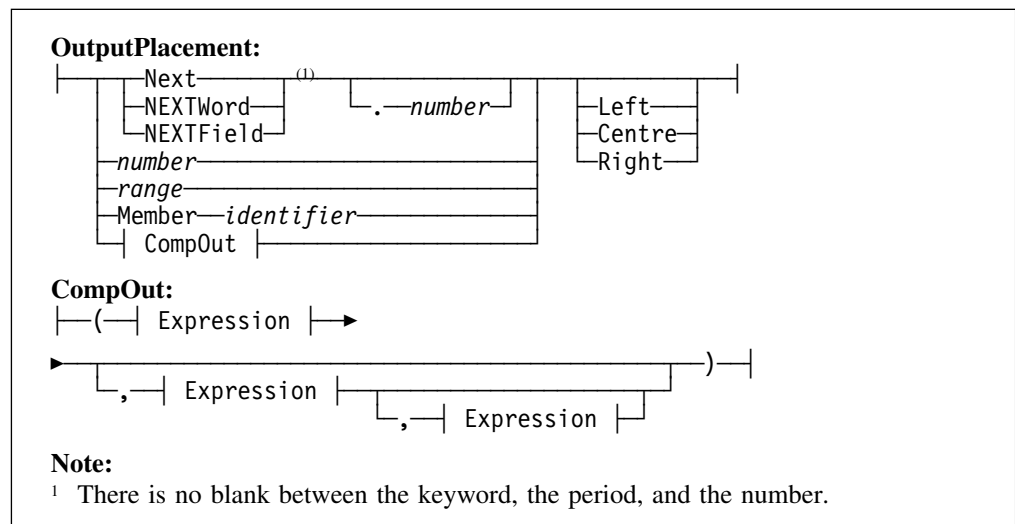
Figure 404. Composite Conversions Supported

	D	X	B	F	V	P	I	T	U
D		D2X	D2B						
X	X2D		X2B	X2F	X2V	X2P	X2I	X2T	X2U
B	B2D	B2X		B2F	B2V	B2P	B2I	B2T	B2U
F		F2X	F2B						
V		V2X	V2B						
P		P2X	P2B						
I		I2X	I2B						
T		T2X	T2B						
U		U2X	U2B						

Composite conversion (x2y) is performed strictly via the C format; that is, x2C followed by C2y.

Output Placement

The output placement specifies where a field is stored in the output record. It consists of a position and a placement option, which specifies the alignment of the data within the field. When no explicit length is specified for the output field, the length of the data to be stored is used as the size of the output field.



- NEXT Append the field to the end of the output record built so far. You may append a period and a number to specify an explicit field length.
- NEXTWORD Append a blank to the output record if it is not null. Then append the field to the end of the output record built so far. You may append a period and a number to specify an explicit field length. NWORD is a synonym; it can be abbreviated to NW.

NEXTFIELD NF	Append a horizontal tabulate character (X'05') to the output record if it is not null. Then append the field to the end of the output record built so far. You may append a period and a number to specify an explicit field length. NFIELD is a synonym; it can be abbreviated to NF.
<i>number</i>	Specify the beginning column of the output field. The field will be the size of the source data to be stored.
<i>range</i>	Specify the extent of the output field. If the field length is different from the size of the data to be stored, the data are padded with the pad character or truncated on the right, unless a placement option is specified.
MEMBER	Specify the identifier for the member that defines the output field and its type. For members that are typed D, F, or P and without picture, explicit conversion, or STRIP, the input source is converted automatically to the requested output type. When the input is not a typed member, the character input field is converted to a counter and thence to the requested output format. Direct automatic conversion between the three types is also applied when the input source is a typed field and no explicit conversion or STRIP is specified, except that conversion to or from packed decimal is performed via assigning the value to a counter, which may truncate the significant digits to 31.
Expression	<p>The parenthesised expression specifies the column number and optionally the field width and placement. The first expression must evaluate to a positive integer; the second expression must evaluate to an integer that is zero or positive; and the third expression must evaluate to a string that matches one of the placement options described below. A length of zero specifies that the length is the default length for the particular input source.</p> <p>When the third expression is present, the placement option described below is ignored.</p> <p>The parentheses are required.</p> <p>These two output placements are identical, except that the second one will take somewhat longer than the first one:</p> <pre>... 1.8 c (1, 8, "c") ...</pre>

An optional keyword specifies the placement of the source field within the output field; this is called a *placement option*. When a placement option is specified, the input field after conversion (and thus after the default length of the output field is determined) is stripped of leading and trailing blank characters unless the conversion is D2C, F2C, I2C, P2C, U2C, or V2C. This field is then inserted in the output field, truncated or padded with the pad character, as specified by the keyword used:

LEFT The field is aligned to the left of the output field, truncated or padded on the right with pad characters.

- CENTRE The field is loaded centred in the output field truncated or padded on both sides with pad characters. If the field is not padded equally on both sides, the right side gets one more pad character than the left side. If the field is not truncated equally on both sides, the left side loses one more character than the right side.

 CENTER is also recognised.
- RIGHT The field is aligned to the right of the output field truncated on the left or padded on the left with pad characters.

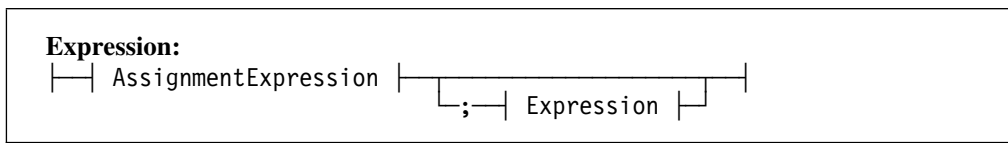
Expression

: An expression contains terms that are combined with operators. All REXX numeric operators are supported, except for the exponentiation operator (**) and the exclusive OR operator (&&). The REXX concatenate operator (|) is also supported, but the blank operator is not (concatenate with blank). In addition, several operators are borrowed from the C language, as is the notion that assignment is an operator.

A few diagrams are required to show the correct precedence of the conditional operator, which selects one of two expressions depending upon the result of evaluating a third expression. The precedence of the remaining operators is not shown by diagrams, but by the order in which they are described.

Blanks are ignored between syntactic entities in the part of an expression that is enclosed in parentheses. But an expression that contains no parenthesis cannot contain blanks, because the first blank will mark the end of the expression. Enclose the entire expression in parentheses to be able to use blanks liberally.

: *spec* parses expressions differently than most other components of *CMS Pipelines*. For example, a *word* need not be blank-delimited (in some contexts, it must not be followed by a blank); an operator or a separator will do just as well, while in other stages or not within an expression, the operator or separator would be included in the word.

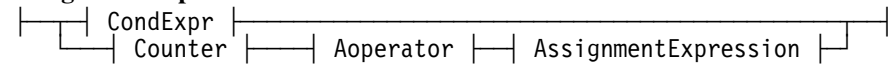
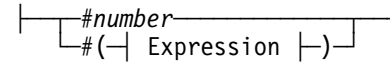


The semicolon operator evaluates its left hand operand and then its right hand one. The result is the left hand operand; the right hand result is discarded. Thus, the semicolon operator can be used to reset a counter after it is printed.

Assignment Expression

The operand of the semicolon operator can be a conditional expression (which is defined later); or it can be an assignment expression.

An assignment expression evaluates the first operand and then assigns a value to the specified counter. How this is done depends on the particular assignment operator used.

AssignmentExpression:**Counter:**

After the pound sign, specify the number of the counter to receive a value or an integer expression in parentheses to compute the number of the counter.

The assignment operators are:

:= Assignment. The counter is assigned the value of the right hand side.

Nota bene: A colon is used to distinguish the assignment operator from the relational equality operator.

+= Increment. The right hand operand is added to the counter.

-= Decrement. The right hand operand is subtracted from the counter.

***=** The counter is multiplied by the right hand operand.

/= The counter is divided by the right hand operand.

%= The counter is divided by the right hand operand. The result is truncated to an integer.

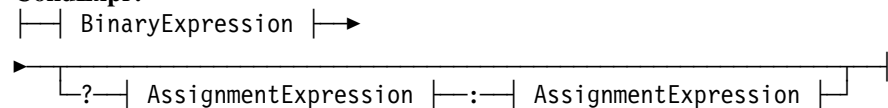
//= The counter is assigned the remainder after division by the right hand operand.

$x//y == x - ((x\%y)*y)$

||= The string representation of the right hand operand is appended to the contents of the counter, which is converted to a string, as required.

Conditional Expression

The conditional expression evaluates a binary expression. The question mark operator is a ternary operator. It evaluates its first operand and then one of its two remaining operands, which are separated by a colon. When the first operand evaluates true, the operand to the left of the colon is evaluated and the other one is ignored; when the result is false, the colon's left operand is ignored and the right hand one is evaluated.

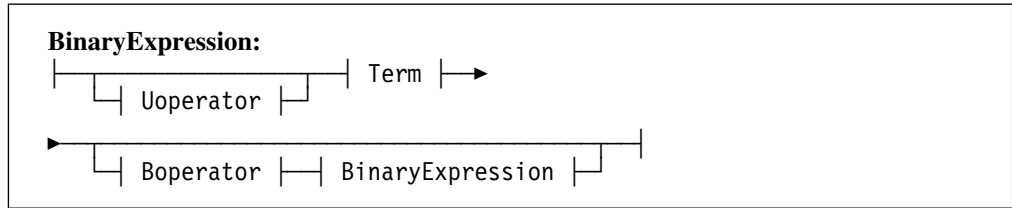
CondExpr:

The two assignment expressions must be of the same type. The result takes on this type.

Two field identifiers are taken to be numeric; one field identifier is taken to be a string reference when the other expression is a string expression. The built-in function `string()` can be used to cast a field identifier into a string.

When one of the assignment expressions is a string and the other one is numeric, that number will be converted to a string.

Binary Expression



The unary operators bind closest to a term. They are:

- Minus. The sign is inverted.
- + Plus. The term is unchanged.
- ~ Negate. When the term is zero, the result is the number 1. Otherwise the result is the number 0.

The binary operators are described here in the order of precedence, the operators that have the highest precedence are first. Operators in a group have the same precedence. Operators that have the same precedence group from left to right.

: Operands are converted to the appropriate type, as required. An error is reported when a
: string operand cannot be converted to a number.

: Note that unlike REXX, numeric comparison operators do not perform string comparisons
: when an operand is not numeric.

When the order of evaluation of the operands is not specified, it is indeed *unspecified*. Do not rely on the order of evaluation, even though you can determine it easily enough. For example, it is unpredictable whether an assignment in one operand has effect for the evaluation of the other operand.

The multiplicative operators have the highest precedence of the binary operators.

- * Multiplication.
- / Division.
- % Integer division. Truncate the result to an integer.
- // Remainder after division.
 $x//y == x - ((x\%y) * y)$

Addition and subtraction.

- + Add.
- Subtract.

: The concatenate operator is alone in its group.

: || Concatenate the two string operands.

The relational operators. The result is 1 if the relation holds; otherwise it is 0.

- : < Test for the first operand being numerically less than the second one.
- : <= Test for the first operand being numerically less than or equal to the second
: one.
- : >

:	>	Test for the first operand being numerically greater than the second one.
:	>=	Test for the first operand being numerically greater than or equal to the second one.
:	<<	Test for the first operand being string being strictly less than the second one.
:	<<=	Test for the first operand being string being strictly less than or equal to the second one.
:	>>	Test for the first operand being string being strictly greater than the second one.
:	>>=	Test for the first operand being string being strictly greater than or equal to the second one.

Equality operators. The result is 1 if the relation holds; otherwise it is 0.

:	=	Test for the first operand being numerically equal to the second one.
:	!=	Test for the first operand being numerically not equal to the second one.
:	==	Test for the first operand being string being strictly equal to the second one.
:	!=	Test for the first operand being string being strictly not equal to the second one.

The AND operator is alone in its group.

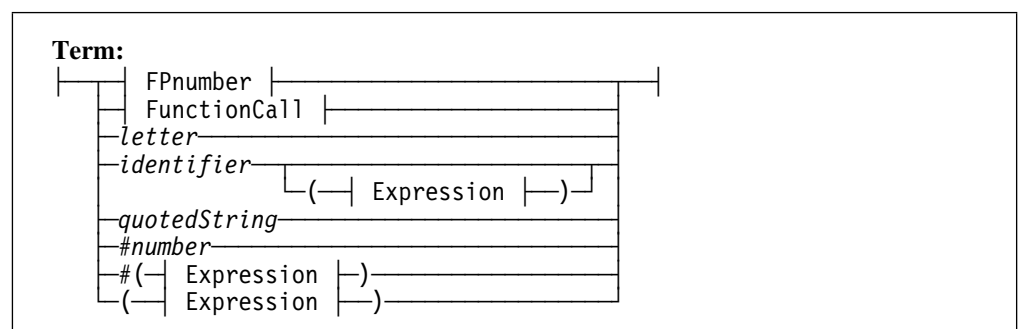
& First evaluate the left hand side. When it evaluates to 0, the result is set to 0 and the second operand is not evaluated. The second operand is evaluated only if the first one evaluates to a nonzero value. If the second operand then evaluates to 0, the result is set to 0. The result is 1 only when both operands evaluate to a nonzero value. Note that this behaviour is different from REXX; it is similar to the C && operator.

The OR operator is alone in its group.

| First evaluate the left hand side. When it evaluates to a nonzero value, the result is set to 1 and the second operand is not evaluated. The second operand is evaluated only if the first one evaluates to 0. If the second operand then evaluates to a nonzero value, the result is set to 1. When both operands evaluate to 0, the result is 0. Note that this behaviour is different from REXX; it is similar to the C || operator.

Term

A term represents a floating point number, the value of an identified field, the value of a counter, the result of a call to a built-in function, the contents of a member of a structure, or the result of evaluating an expression in parentheses.



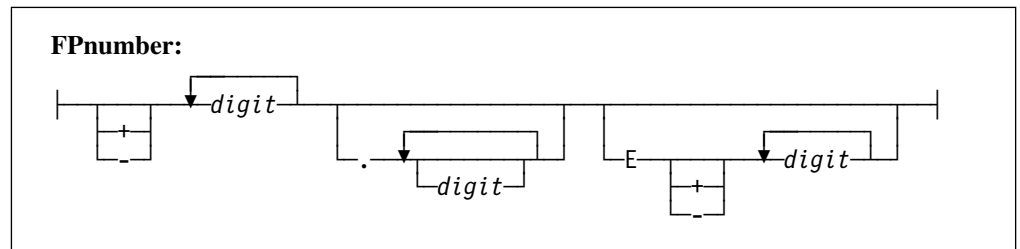
<i>letter</i>	Specify the identifier for a field to reference the contents of that field. When used alone in assignment and PRINT and no type is specified, the contents of the field are converted to the internal representation of a number. When no specification item has been declared for the letter, it is parsed as an <i>identifier</i> instead.
<i>identifier</i>	Specify the name of a member of a structure, optionally with a computed subscript. The identifier may specify a fully qualified member name by prefixing it with two periods; you may specify that the current qualifier is to be used by prefixing one period. When the structure contains nested structures, such structures can also be subscripted. While the subscript is shown as optional, it should be taken to mean that you must specify a subscript when referencing a member that is an array; and you may not specify a subscript for a scalar member. Note in particular that this usage does not apply to an <i>inputRange</i> that specifies a field in the input record; for that you are limited to constant subscripts. When the identifier resolves to a member of a structure that is a manifest constant, the value of the constant is used as if it were entered as a number. Thus <code>vmcparm.vmcpsend</code> will resolve to the number 2.
<i>#number</i>	Specify the number of the counter to reference.
<i>#()</i>	Specify an expression to compute the number of the counter to reference.
<i>quotedString</i>	Specify a character string for strict comparison operators. The string follows the REXX rules. That is, double occurrences of the enclosing quote specify a single occurrence of the enclosing quote inside the string; hexadecimal constants are denoted by an X after the closing quote. Binary constants are denoted by a B after the closing quote. '0', 'f0'x, and 1111000'b all designate the same character.

The ambiguity in the syntax of counters, field identifiers, functions, and member names is resolved as follows:

- A single letter that has also been specified as an identifier for an input source is scanned as a field identifier.
 - A number sign (#) that is followed by only digits and no letters or any of the special characters @#\$! is scanned as a counter.
 - An identifier followed by a left parenthesis is scanned as a function name when it represents one of the built-in functions or a user written function in a filter package.
Otherwise it is scanned as a subscripted member. You must prefix a member name that is also a built-in or user written function by a period to select the active qualifier.
 - Otherwise it is scanned as an *identifier*. Thus #0x is scanned as an *identifier*.
- Note that identifiers in expressions cannot contain a question mark because that is scanned as the conditional operator, but they are valid in input sources and output placements.

Floating point Numbers

A number consists of an optional sign followed by an integer part. A fraction and an exponent part are optional. Blanks are not allowed in a floating point number. This syntax also applies to input fields that are converted to the internal representation of numbers.



The implementation also supports fractional numbers that begin with a period; for example, .5. In this format, at least one digit must be specified after the period. To retain clarity, this is not shown in the railroad track above.

Any additional significant digits beyond the 31-digit precision are ignored.

Functions

spec supports user written functions in type-2 filter packages, as well as built-in ones. Some, but not all, built-in functions can be replaced by functions in the PTF filter package; which ones is unspecified and may change over time. The search order is:

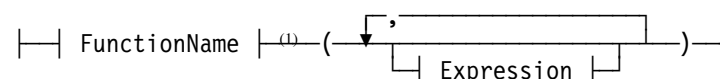
1. “Hardwired” function names.
2. Functions in the PTF filter package.
3. Other built-in functions (those that are not hard wired).
4. Functions in other filter packages, in the order they were loaded.

Built-in Functions

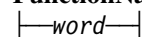
spec expressions use a number of built-in functions. They are described here in four sections:

- Functions that perform as the function by the same name in the REXX language.
- Functions that are particular to spec and return a Boolean value.
- Functions that are particular to spec and return a number.
- Functions that are particular to spec and return a string.

FunctionCall:



FunctionName:



Note:

¹ No blanks are allowed between the name and the opening parenthesis.

While arguments are, in general, expressions, some functions require particular data, such as a single letter. This is noted in the syntax diagrams below.

Functions Modelled on REXX

The following functions have the same argument requirements and results as the corresponding REXX function.

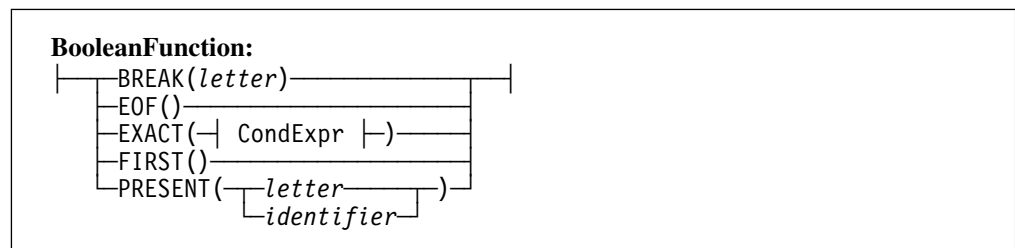
The functions are not described further (except for word, which is enhanced); refer to REXX documentation for details.

ABBREV	C2X	LEFT	SPACE	WORDPOS
ABS	DATATYPE	LENGTH	STRIP	WORDS
BITAND	DELSTR	MAX	SUBSTR	XRANGE
BITOR	DELWORD	MIN	SUBWORD	X2C
BITXOR	FIND	OVERLAY	TRANSLATE	X2D
CENTRE	INDEX	POS	VERIFY	
COMPARE	INSERT	REVERSE	WORD	
COPIES	JUSTIFY	RIGHT	WORDINDEX	
C2D	LASTPOS	SIGN	WORDLENGTH	

Notes:

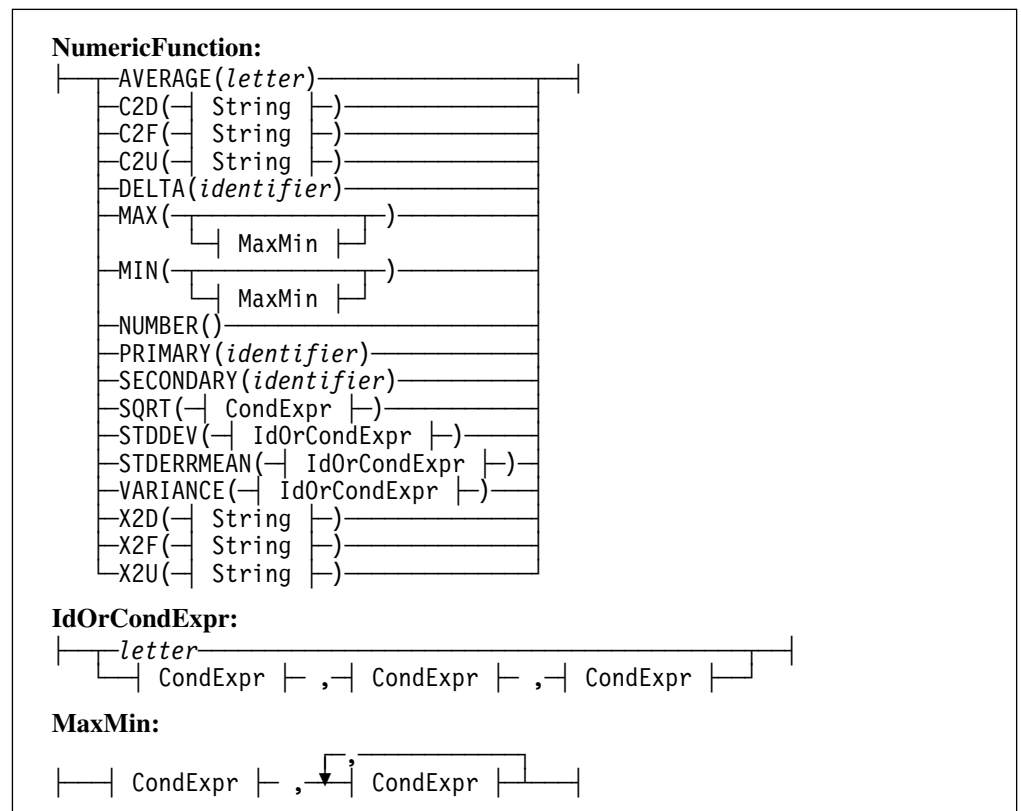
1. datatype with one argument returns NUM when the first string can be assigned to a counter without error. Thus, it returns NUM for many strings that cannot be processed by the D2C conversion on a specification item.
2. Several conversion functions by the same name as a REXX function are defined as specific to *spec* because they either perform a slightly different function or they support fewer operands.
3. max and min with no or two or more arguments are described below.
4. word is also described below, because it supports a third argument to make it parallel to field.
5. *spec* has no concept of NUMERIC DIGITS, which may cause the numeric result of a function to be converted to string differently from REXX.

Boolean Built-in Functions



break	The argument must be a single literal letter, which specifies a field identifier that must have been associated with an input field previously in the specification list. <code>break()</code> returns 1 if a break has been established on the level specified. On the runin cycle, no break is reported if the field is null. A break is always reported on the runout cycle.
eof	A niladic function. <code>eof()</code> returns 1 during the runout cycle. It returns 0 during all other cycles.
exact	The argument must evaluate to a number. Often it is just a term that refers to a counter. The result is 1 if no truncation has occurred in the evaluation of the expression. The estimate is conservative, that is, the result may be zero even when the argument is in fact exact.
first	A niladic function. <code>first()</code> returns 1 during the runin cycle. It returns 0 during all other cycles.
present	Return 1 when the field is present in the record and 0 otherwise. <code>length()</code> for the contents of a field is 0 both when the field is present, but null, and when the field is not present.

Numeric Built-in Functions

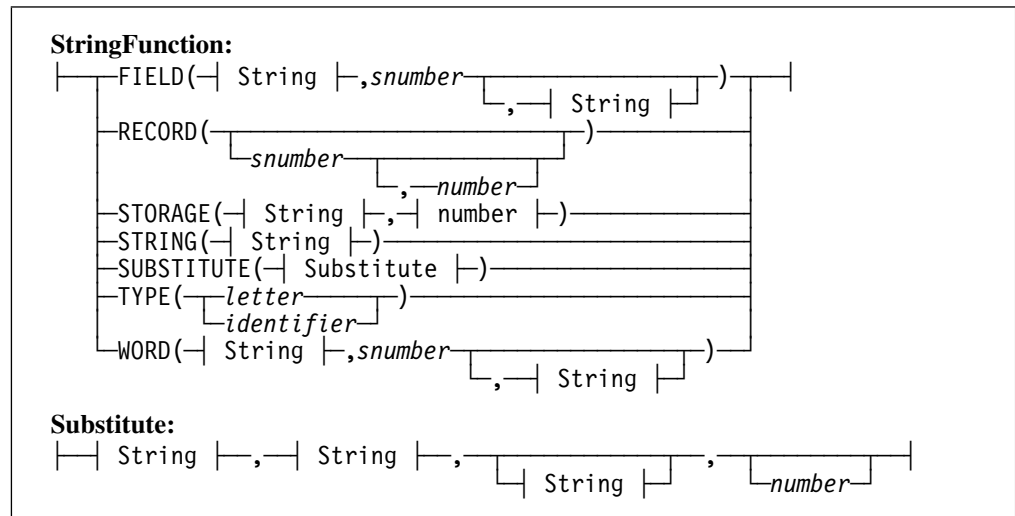


spec Reference

.	average	The argument must be a single literal letter, which specifies a field identifier that must have been associated with an input field previously in the specification list. The contents of the specified field must evaluate to a number in all input records. The number returned is the average over the input records seen so far. The average of no records is zero.
.	c2d	The argument is a string expression. The string is interpreted as a binary number in two's complement notation. The maximum number of significant input bits is 108 (which is not an integral number of bytes). The result is a number that expresses the binary input number in decimal.
:	c2f	The argument is a string expression of two to sixteen bytes. The string is interpreted as the internal representation of a System/360 hexadecimal floating point number.
:	c2u	The argument is a string expression. The string is interpreted as an unsigned binary number. The maximum number of significant input bits is 108 (which is not an integral number of bytes). The result is a number that expresses the binary input number in decimal.
!	delta	Returns the difference in value of a member of a structure in subsequent input records. A convenience for <code>primary(x)-secondary(x)</code> . The second term is zero during the runin cycle, effectively making the result equal to the first term. Use <code>if first() then noprint else .. endif</code> to avoid working with the full value from the first record during the runin cycle. delta implies both <code>SELECT FIRST</code> and <code>SELECT SECOND</code> .
.	max	None, two, or more arguments must be specified. All arguments must evaluate to numbers. The result is the largest number in the argument list. When no arguments are specified, <code>max()</code> returns the largest (most positive) number that a counter can store.
.	min	None, two, or more arguments must be specified. All arguments must evaluate to numbers. The result is the smallest number in the argument list. When no arguments are specified, <code>min()</code> returns the smallest (most negative) number that a counter can store.
.	number	<code>number()</code> is niladic. The result is the number of the current cycle, starting with 1 on the runin cycle, if any is taken. During the runout cycle <code>number()</code> returns the total number of cycles taken. That is, it is not incremented during the runout cycle, unlike the <code>NUMBER</code> data source.
:	primary	Return the numeric contents of the specified member from the first reading. The member must have a type that is D or F. <code>primary</code> implies <code>SELECT FIRST</code> .
:	secondary	Return the numeric contents of the specified member from the second reading. The member must have a type that is D or F. <code>secondary</code> implies <code>SELECT SECOND</code> .
.	sqrt	The argument must evaluate to a number that is zero or positive. The result is the square root of the number.

.	stddev	<p>For the monadic <code>stddev()</code> the argument must be a single literal letter, which specifies a field identifier that must have been associated with an input field previously in the specification list. The contents of the specified field must evaluate to a number in all input records. For the triadic <code>stddev()</code> the arguments must all evaluate to numbers. The first argument is considered to be the sum of a series of numbers (s); the second argument is considered to be the sum of the squares of the series (q) and the third argument is considered to be the count of observations (n). <code>stddev(f)</code> evaluates the triadic <code>stddev(s, q, n)</code> on the values seen so far in the field f.</p> $\text{stddev}:=\text{sqrt}(\text{variance}(s, q, n))$
.	stderrmean	<p>For the monadic <code>stderrmean()</code> the argument must be a single literal letter, which specifies a field identifier that must have been associated with an input field previously in the specification list. The contents of the specified field must evaluate to a number in all input records. For the triadic <code>stderrmean()</code> the arguments must all evaluate to numbers. The first argument is considered to be the sum of a series of numbers (s); the second argument is considered to be the sum of the squares of the series (q) and the third argument is considered to be the count of observations (n). <code>stderrmean(f)</code> evaluates the triadic <code>stderrmean(s, q, n)</code> on the values seen so far in the field f.</p> $\text{stderrmean}:=\text{stddev}(s, q, n)/\text{sqrt}(n-1)$
.	variance	<p>For the monadic <code>variance()</code> the argument must be a single literal letter, which specifies a field identifier that must have been associated with an input field previously in the specification list. The contents of the specified field must evaluate to a number in all input records. For the triadic <code>variance()</code> the arguments must all evaluate to numbers. The first argument is considered to be the sum of a series of numbers (s); the second argument is considered to be the sum of the squares of the series (q) and the third argument is considered to be the count of observations (n). <code>variance(f)</code> evaluates the triadic <code>variance(s, q, n)</code> on the values seen so far in the field f.</p> $\text{variance}:=q/n-(s/n)**2$
:	x2d	Similar to <code>c2d</code> , except the the input is an unpacked hexadecimal representation of the signed two's complement binary number.
:	x2f	Similar to <code>c2d</code> , except the the input is an unpacked hexadecimal representation of the hexadecimal floating point number. (Two different meanings of "hexadecimal".)
:	x2u	Similar to <code>c2d</code> , except the the input is an unpacked hexadecimal representation of the unsigned binary number.

String Built-in Functions



The string argument may be a literal letter which refers to the identified input range; an identifier that specifies a member; a quoted string literal; or the result of a string function.

- field** Return the nth field of the first string. The number must be nonzero. The field number is counted from the end of the string when the number is negative. If present, the second string argument must have length 1; it specifies the field separator; the default is X'05', horizontal tab.
- record** Return the current input record or a substring of it. If present, the first number specifies the starting position of the result within the record; it must be nonzero. When this number is negative, the position is relative to the end of the record. The second number specifies the maximum number of columns to include in the result. The default is to the end of the record. The second number must be zero or positive. The result is never padded.
- storage** Return the contents of virtual storage at the specified address.

The first argument is the address in printable hexadecimal; the second argument is a number that must be zero or positive.

```

pipe spec eof print storage('230', 32) 1 | console
▶VM Conversational Monitor System
▶Ready;

```
- string** The result is the argument string without modification, syntactically cast as a string. Thus, string("1") returns a string, not a number. string() may be needed in conjunction with the conditional operator to cast one of the operands into a string.
- substitute** Return a modified string where occurrences of one substring are replaced by another, much as done the the XEDIT change command. The first argument is the string to be changed. The second argument is the substring to be replaced; it must be at least one character. The third argument is the replacement string, the default is an empty string. The fourth argument, if present, must be positive. It specifies the maximum number of substitutions to perform; the default is infinity.

:	type	Return the data type associated with the field (if it is a member of a structure) or member, or a single blank.
:		
:	word	Return the nth word of the first string. The number must be nonzero. The word number is counted from the end of the string when the number is negative. If present, the second string argument must have length 1; it specifies the word separator; the default is X'40', a blank.
:		
:		
:		

Pictures

A *picture* is a pictorial description of the desired formatting of a numeric quantity. It is specified after the keyword PICTURE.

A picture contains a character for each column of the formatted field, except that the V character does not represent an output character. Thus, the output fields are fixed in length. Case is ignored in pictures.

The picture characters are a subset of the ones defined for PL/I. They are S+-\$9Z*Y,./BVE. They comprise five groups: Sign, digit select, punctuation, implied decimal point, and exponent.

It is customary to *suppress leading zeros* in formatted numbers. To support this, *spec* maintains a significance latch internally. (“Latch” is the engineer’s jargon for what a programmer calls a “switch”.) The latch remembers that a significant digit has been met.

A leading zero is replaced with a blank when the significance latch is off and the picture character is the letter “Z”. The significance latch is off at the beginning of the picture. It is turned on by a nonzero digit or the picture character 9. It is forced off again by the E pattern character.

Sign Characters

S	(The letter S.) Insert the sign (+ or -) of the number. Zero is considered positive.
+	Insert a + if the number is zero or positive; a blank if the number is negative.
-	Insert a - if the number is negative; a blank if the number is not negative.
\$	(The dollar sign.) Insert the currency symbol irrespective of the sign of the number. (This is the pound sterling symbol on a UK terminal.)

A sign character can be stand-alone or part of a *drifting sign*. A stand-alone sign character occupies the column where it is specified. A drifting sign is specified by successive columns containing the particular sign character, possibly with interspersed punctuation characters. The sign occupies the column before the one where the significance latch is turned on. Prior columns contain a blank. The first column of a drifting sign can never contain a digit.

Digit Selection

9	Insert a digit. The significance latch is set unconditionally.
Z	Insert a digit, suppressing leading zeros. The position contains a blank when the digit is zero and the significance latch is off. It contains a digit if the digit is nonzero or the significance latch is on. A nonzero digit sets the significance latch on.
*	Insert cheque protection. The position contains an asterisk when the digit is zero and the significance latch is off. It contains a digit if the digit is nonzero or the significance latch is on. A nonzero digit sets the significance latch. That is, the asterisk is similar to Z, except that it inserts an asterisk rather than a blank when it processes a leading zero.
Y	Insert a blank when the digit is zero, and the digit otherwise. A nonzero digit sets the significance latch. Thus, Y is similar to Z, but it does not test the significance latch.

Punctuation

,	Insert a comma if the significance latch is set; insert a blank otherwise.
.	Insert a period if the significance latch is set; insert a blank otherwise.
/	Insert a forward slash if the significance latch is set; insert a blank otherwise.
B	Insert a blank.

Implied Decimal Point

V	Insert nothing. The position corresponds to the implied beginning of the fraction of the number. The significance latch is set if the counter contains a nonzero value. When no V is present in a picture, a period (if one is present) indicates the implied beginning of the fraction.
---	--

! The implied V in the picture is not compatible with the picture in PL/I
! and the use of this feature is discouraged.

Exponent

E	Insert the character "E". This marks the beginning of the exponent. The exponent field can contain only an optional sign, which cannot drift, followed by digit selectors Z (suppress zero) or 9.
---	---

General

The default picture is -----9, which is eleven columns with a drifting minus sign.

These rules apply to the pattern as a whole:

1. There can be at most one E and its attendant exponent field.
2. There can be at most one V. The V must be before the E (if one is present).
3. A sign that does not drift can be at the beginning of the picture or after the decimals. Only an exponent may follow a sign that is after digit selectors.
4. A drifting sign must be specified with the same sign character in all positions.

5. Drifting signs, zero suppress, and cheque protection are mutually exclusive. Only the digit selector 9 can be used after any of these.

Continental European Conventions

Pictures can be used to format numbers according to Continental European conventions, for example `zzz.zz9v,99`. Here, the `V` is mandatory. If it were omitted, it would be assumed that the fraction begins after three digits rather than six. Note that a number formatted in this way cannot be processed directly by another *spec* stage. You can use *change* to remove the periods and *xlate* to change the comma to a decimal point.

Chapter 25. Pipeline Commands

This chapter contains Programming Interfaces.

The default command environment of a REXX program running as a pipeline filter processes pipeline commands, described in alphabetical order in the following sections. This list is an overview by function.

Figure 405. Overview of Pipeline Commands

Transport Data	BEGOUTPUT	Enter implied output mode.
	READTO	Read a record from the currently selected input stream.
	PEEKTO	Preview the next record from the currently selected input stream. The record stays in the pipeline.
	OUTPUT	Write the argument string to the currently selected output stream.
	SHORT	Connect the currently selected input stream to the currently selected output stream. The streams are no longer connected to the program.
Control Pipeline Topology	ADDPIPE	Add a pipeline specification to run in parallel with your program. This is used, for instance, to replace the current input stream with a file to embed.
	ADDSTREAM	Add an unconnected stream to the program.
	CALLPIPE	Run a subroutine pipeline.
	SELECT	Select an input or output stream, or both. Subsequent requests for data transport are directed to this stream.
	SEVER	Detach a stream from the program. The other side of the connection sees end-of-file.
Control Programs	COMMIT	Commit the program to a particular level. The return code is the aggregate return code for the pipeline specification so far.
	EOFREPORT	Modify the return codes reported by the data transport commands.
	NOCOMMIT	Disable automatic commit first time a program performs an I/O operation.
	REXXCMD	Call a subroutine REXX program. The program has access to the pipeline command environment and the caller's streams.
	SETRC	Set the return code for the program writing a record to the currently selected input stream.
Scanning	SUSPEND	Allow other stages to run.
	GETRANGE	Extract part of record or string.
	SCANRANGE	Parse an <i>inputRange</i> .
Issue Messages	SCANSTRING	Parse an <i>delimitedString</i> .
	MESSAGE	Write the argument string as a message.
Query Program's Environment	ISSUEMSG	Issue a <i>CMS Pipelines</i> message. The argument specifies a message number; the message text is obtained from a message table.
	MAXSTREAM	Return the highest stream number available.
	RESOLVE	Return entry point address for a pipeline program.
	STAGENUM	Return position in pipeline.
	STREAMNUM	Return stream number corresponding to an identifier.
	STREAMSTATE	Return connection status of stream.

The following pipeline commands are also available to the *pipcmd* built-in program and the underlying macro PIPCMD: ADDPIPE, ADDSTREAM, CALLPIPE, COMMIT, EOFREPORT, ISSUMSG, MAXSTREAM, MESSAGE, OUTPUT, RESOLVE, REXXCMD, SELECT, SETRC, SEVER, SHORT, STAGENUM, STREAMNUM, STREAMSTATE, SUSPEND. Whether it makes sense to use them all with *pipcmd* is another matter; in particular, COMMIT to a positive number will cause a stall.

The following pipeline commands are available only through the REXX interface: BEGOUTPUT, GETRANGE, NOCOMMIT, PEEKTO, READTO, SCANRANGE, SCANSTRING, and STREAMSTATE ALL.

Return code -7 on a pipeline command means that the pipeline command processor cannot resolve the command. Refer to “Return Codes -3 and -7” on page 116.

ADDDPIPE—Add a Pipeline Specification to the Running Set

```
▶▶—ADDDPIPE—| pipeSpec |—▶▶
```

Syntax: The argument string to ADDPIPE is processed as a pipeline specification. Refer to Chapter 21, “Syntax of a Pipeline Specification Used with PIPE, *runpipe*, ADDPIPE, and CALLPIPE” on page 237.

Operation: The pipeline specification is added to the current set of pipelines. Its stages run in parallel with the stage issuing ADDPIPE, independent of the commit level of the stage that issues the ADDPIPE pipeline command.

Connectors in the pipeline specification designate how the stage’s current streams are modified. All streams mentioned in connectors are disconnected from the stage.

- The stream is connected to the new pipeline when the connector is before a pipeline and the second component of the connector is INPUT, or the connector is after the pipeline and the second component is OUTPUT.

```
/* Process input and output independently */
"addpipe *.input:|xlate upper|> output file a"
```

The stage cannot reconnect to a stream that has been transferred to another pipeline in this way.

- The connection is saved on a stack for the stream when the the connector is after a pipeline and the second component is INPUT, or the connector is before the pipeline and the second component is OUTPUT. The new pipeline is connected to the stream instead of the saved connection. End-of-file on the new connection sets return code 12 in a READTO or PEEKTO pipeline command. SEVER restores the stacked connection.

```
/* Read parameter file */
"addpipe < parm file | *.input:"          /* Connect input to file      */
"nocommit"                                /* Disable automatic commit   */
"readto line"                              /* Read first line of file    */
do while RC=0                               /* Process all lines          */
  /* Process line */
  "readto line"                             /* Read next line            */
end
"sever input"                               /* Reinststate input file     */
"commit 0"                                  /* See if other stages are OK */
if RC/=0 then exit 0                       /* Exit quietly if not       */
```

ADDSTREAM • BEGOUTPUT

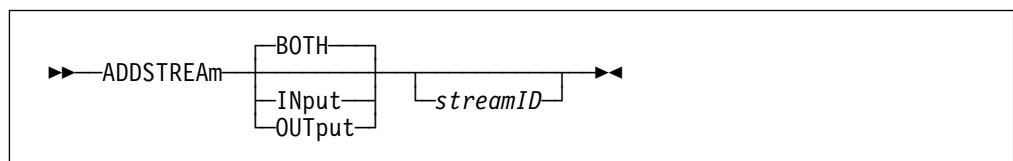
- A pipeline is inserted in front of (or after) the stream when a side of a stream is referenced both at the beginning and end of a pipeline, for instance:

```
"addpipe *.input: |deblock net|*.input:"  
"addpipe *.output: |xlate upper|*.output:"
```

Return Codes:

- 0 The pipeline specification has been added to the running set. The stage issuing ADDPIPE runs in parallel with the stages added to the pipeline set.
- ± The pipeline specification has one or more syntax errors. Error messages are issued.

ADDSTREAM—Create a Stream



Syntax: ADDSTREAM accepts an optional keyword and an optional stream identifier.

Operation: An unconnected stream is added to the stage on the side(s) specified by the first keyword. If present, the stream identifier is set; the stream has no identifier by default. Use MAXSTREAM to discover the number of the stream just added. Use ADDPIPE to connect a stage to the stream.

Return Codes:

- 0 The stream(s) are added to the stage.
- 112 Too many arguments. This may be caused by a misspelled keyword that is assumed to be a stream identifier.
- 174 A stream already exists with the stream identifier specified.

Examples:

```
"addstream output errs"  
"addpipe *.output.errs:|> error file a"
```

Remember to select the error stream before using OUTPUT to write to it. Select the primary output stream when writing “normal” output.

BEGOUTPUT—Enter Implied Output Mode



Syntax: A string is optional with BEGOUTPUT. It specifies an ending delimiter record. A null delimiter record is assumed when the string is omitted. The string is truncated after eight bytes.

Operation: The argument string is stored, but is otherwise ignored; the REXX interface then enters the implied output mode.

Subsequent pipeline commands are treated as output data rather than commands. That is,

they are processed by the OUTPUT pipeline command; this includes implied commit processing. The complete command is written to the currently selected output stream.

A command that contains exactly the string specified (or is null when no string was specified) terminates the implied output mode and the command interface reverts to its normal operation; the terminating command is discarded.

Return Codes: The return code is always 0.

Examples: To write two lines of output. Note that the lines are still processed as REXX expressions. In practice this means that literals must be enclosed in quotes.

```
'begoutput'
'Field 1      Field 2'
'-----'
''
'callpipe *: | spec 37.14 1 89-* 16 | *:' /* Command mode active */
/* Null command to terminate */
```

Notes:

1. BEGOUTPUT can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.

CALLPIPE—Run a Subroutine Pipeline

▶▶—CALLPIPE—| pipeSpec |—▶▶

Syntax: The argument string to CALLPIPE is processed as a pipeline specification. Refer to Chapter 21, “Syntax of a Pipeline Specification Used with PIPE, *runpipe*, ADDPIPE, and CALLPIPE” on page 237.

Operation: The subroutine pipeline may be connected to the stage’s input and output streams. The stage issuing CALLPIPE is suspended until all stages of the subroutine pipeline have returned. The stage that issues CALLPIPE commits to the highest commit level of the subroutine pipeline while it waits for it to complete. The subroutine pipeline can run on a commit level that is lower than the caller’s. A short-through pipeline forces a commit to level 0 to avoid a stall.

Connectors in the subroutine pipeline connect to streams in the stage issuing the CALLPIPE pipeline command until end-of-file is transferred across the connection; the connection to the calling stage is restored when end-of-file is transmitted from the subroutine.

Return Codes:

- 0 The pipeline specification is syntactically correct. All stages of the pipeline return code 0.
- ± There is a syntax error in the pipeline specification or a stage of the subroutine pipeline gives a return code that is not zero.

Examples: This subroutine pipeline takes a literal, makes it upper case, and passes it on the currently selected output stream.

COMMIT

TCALLP REXX	Example of Use
<pre>/* CALLPIPE example */ 'callpipe (name TCALLP)', ' literal Hello, world', ' xlate upper', ' *:' exit RC</pre>	<pre>pipe rexx tcallp console */ ▶HELLO, WORLD ▶Ready;</pre>

To position the input stream at the next line with a comma in column 1 (or read to end-of-file):

```
'callpipe *:||to|label ,|hole'
'peekto'
if RC=12 then exit /* End-of-file */
```

Notes:

1. A pipeline stall is possible if all these conditions are satisfied:
 - The stage issuing CALLPIPE is on a negative commit level.
 - The subroutine pipeline is connected to both an input stream and an output stream.
 - The subroutine completes without committing to level 0 and without running a program that commits to level 0. A stage that issues the SHORT pipeline command without committing satisfies this condition.

The stage should commit to level 0 before issuing a subroutine pipeline of this nature.

COMMIT—Commit Stage to a New Level

▶▶—COMMIT—*number*—◀◀

Syntax: COMMIT requires a numeric argument.

Operation: Commit to the level specified. When the number is less than or equal to the level the stage is already committed at, the return code is the current aggregate return code for the pipeline specification.

The stage is suspended when the level requested is higher than the level the stage is currently committed at. The stage is suspended until all stages in the pipeline specification (and the caller, if the stage is in a subroutine pipeline) have committed at least to the level the stage requests. The return code is the aggregate return code when all stages are committed to the level specified.

REXX programs begin at commit level -1. The interface commits to level 0 when the stage reads or writes unless a NOCOMMIT pipeline command is issued first.

Return Codes:

- 0 All stages that have returned did so with return code zero.
- 2147483648 The arguments are in error. Message 58, 112, or 113 is issued.
- ± A stage has returned with the return code.

Example: Use COMMIT to test the return code of other REXX programs and those built-in programs that are committed to start on level -1 or before. You can abandon the program if the return code is not zero. In the second example below, *console* is not started because the first stage returns with code 112. (TISSUE REXX is shown on page 758.)

TCOMMT REXX	Example of Use
<pre>/* Test commit */ say 'Tcommit started.' 'commit 0' /* Explicit */ If rc/=0 /* Trouble? */ Then exit 0 'output all is well'</pre>	<pre>pipe rexx tcommit console ▶Tcommit started. ▶all is well ▶Ready;</pre>
	<pre>pipe tissue word tcommit console ▶Tcommit started. ▶Excessive options "word" ▶... Issued from stage 1 of pipeline 1 ▶... Running "tissue word" ▶R(00112);</pre>

EOFREPORT—Enable Reporting of Stream Events



Syntax: EOFREPORT requires a keyword argument.

CURRENT	The original <i>CMS Pipelines</i> behaviour is desired. Stream events are ignored when they do not relate to the currently selected stream at the time of I/O.
ALL	Return code 8 is to be reported on PEEKTO and SELECT ANYINPUT when all output streams are severed.
ANY	Return code 8 is to be reported on PEEKTO and SELECT ANYINPUT when all output streams are severed. Return code 4 is to be reported on OUTPUT, PEEKTO, and SELECT ANYINPUT when any stream is severed. For OUTPUT, return code 4 is reported only if the record was not seen by the following stage.

Operation: The return codes reported by OUTPUT, PEEKTO, READTO, and SELECT ANYINPUT are modified, depending on the specified keyword.

Return Codes:

- 0 End-of-file reporting is set as specified. At least one input stream and one output stream are connected.
- 8 No input stream is connected or no output stream is connected (or no stream at all is connected).
- 111 The word is not a recognised option.
- 112 The argument string is more than one word.
- 113 The argument string is empty.

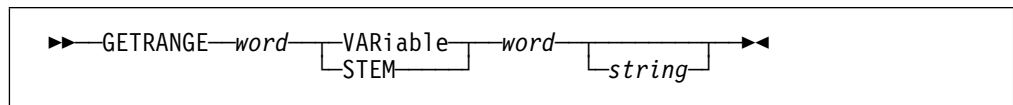
GETRANGE

Example: Use EOFREPORT ALL in stages that should propagate end-of-file. The stage will stop waiting for an input record when the output stream is severed.

Use EOFREPORT ANY. in multistream stages that need to propagate end-of-file immediately.

GETRANGE—Extract Part of Record or String

GETRANGE can be used by the filter programmer to extract the part of a record to be processed in the same way *CMS Pipelines* built-in programs select a substring of the input record.



Syntax: The first word specifies the name of a variable that contains the token representing the *inputRange*. This variable must have been set by a previous SCANRANGE pipeline command. It must not be modified by the REXX program.

The second word is a required keyword. It specifies how the input range should be returned to the program.

VARIABLE	Return contents of the input range in a single variable.
STEM	Return contents of the input range in a stemmed array. The array has one or three variables. Specify a period at the end of the stem to generate REXX compound variables.

The third word specifies the name of the variable or the stem to receive the input range.

The remaining string after exactly one blank is the record from which to extract the contents of an input range.

Operation: When VARIABLE is specified, the third word contains the name of a single variable, which is set to a substring of the input record, as determined by the contents of the token.

The remainder of this section discusses the operation when STEM is specified. The result is stored into a stemmed array; the third word contains the stem to use. The stem would normally end with a period.

When the input range is not present in the record (as opposed to its being of length zero), the compound variables are set as follows:

<i>stem0</i>	1
<i>stem1</i>	The entire input record.

When the input range is present in the record, the compound variables are set as follows:

<i>stem0</i>	3
<i>stem1</i>	The part of the record up to the beginning of the input range.
<i>stem2</i>	The contents of the input range.
<i>stem3</i>	The balance of the record.

See also: SCANRANGE.

Example: This example program writes the reverse of a substring of the input record:

```

/* Getrange sample */
parse arg inputRange

'scanrange required field rest' inputRange
if RC<=0
  then exit RC

'eofreport all'
signal on error
do forever
  'peekto line'
  'getrange field var range' line
  'output' reverse(range)
  'readto'
end

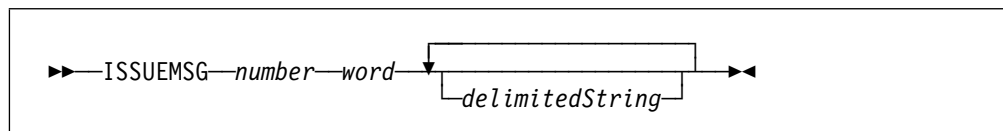
error:
exit RC*(wordpos(RC, '8 12)=0)

```

Notes:

1. GETRANGE can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.
2. The first and the third word are names of variables, but *string* represents a string.

ISSUEMSG—Issue a Message from the Repository



Syntax: The first word of the arguments to ISSUEMSG is the number of the message to issue. The second word should be six characters for the module identifier. Delimited strings are optional after the two required arguments.

Operation: Issue the message with the number specified. The message text is obtained from the internal message text table. There must be a delimited string for each substitution in the message.

See also: MESSAGE. Use MESSAGE to issue messages where you include the message identifier (component, module, number, and severity) as well as the substituted message text.

Return Codes:

- 0 Message 0 is issued.
- + The number of the message issued.
- 58 The first word is not a positive number or zero.
- 60 A delimited string is not properly delimited.
- 113 There are fewer than two words in the argument string.

Example: This example program issues message 112 if the argument string is not blank.

MAXSTREAM

TISSUE REXX	Example of Use
<pre>/* Test ISSUMSG */ If arg(1) = '' Then exit 0 'issuemsg', '112 TSTISSUE /'arg(1)'/ exit RC</pre>	<pre>pipe tissue abc ▶Excessive options "abc" ▶... Issued from stage 1 of pipeline 1 ▶... Running "tissue abc" ▶R(00112);</pre>

Notes:

1. Refer to Chapter 26, “Message Reference” on page 773 for a list of the messages in the built-in message text table.

MAXSTREAM—Return the Highest Stream Number



Syntax: MAXSTREAM requires a keyword to designate which side is queried.

Operation: The return code is set to the highest stream number available on the side specified by the keyword. When a stage starts, it has as many input as output streams defined. Streams can be added to one side with ADDSTREAM.

Return Codes:

- 0 The primary stream is the only stream.
- + The largest number allowed in a SELECT pipeline command.
- 112 The argument string is more than one word.
- 163 No keyword is specified.
- 164 The keyword is not valid.

Example: A program uses MAXSTREAM to test how many streams it has available.

TMAXSTR REXX	Example of Use
<pre>/* Test MAXSTREAM */ 'maxstream output' select when RC=0 Then 'issuemsg 222 TMAXSTR' when RC>1 Then 'issuemsg 264 TMAXSTR' otherwise nop end If RC/=1 Then exit RC</pre>	<pre>pipe tmaxstr ▶Secondary stream not defined ▶... Issued from stage 1 of pipeline 1 ▶... Running "tmaxstr" pipe (end ?) m:tmaxstr?m: ▶Ready; pipe (end ?) m:tmaxstr?m:?m: ▶Too many streams ▶... Issued from stage 1 of pipeline 1 ▶... Running "tmaxstr" ▶R(00264);</pre>

MESSAGE—Issue a Message

```
▶▶MESSAGE—string▶▶
```

Syntax: The argument string to MESSAGE is a substituted message. It should have the standard message identifier in the first ten positions.

Operation: The argument string is issued as a pipeline message.

See also: ISSUMSG issues a message by number using the message text tables built into *CMS Pipelines*.

Return Code: 0.

Example: The message identifier is suppressed when EMSG is set to TEXT so as to display only the text of the message.

TMSG REXX	Example of Use
<pre>/* TMSG REXX Test Message */ 'message tmsgrx001I TMSG here.' exit RC</pre>	<pre>pipe tmsg ▶TMSG here. ▶Ready; cp set emsg on ▶Ready; pipe tmsg ▶tmsgrx001I TMSG here. ▶Ready;</pre>

NOCOMMIT—Disable Automatic Commit on I/O

```
▶▶NOCOMMIT▶▶
```

Syntax: NOCOMMIT accepts no arguments.

Operation: The REXX interface does not commit for level 0 on subsequent I/O commands. To have any effect, NOCOMMIT must be issued before any READTO, PEEKTO, OUTPUT, or SELECT ANYINPUT pipeline commands are issued; otherwise the interface has already committed to level 0.

Return Code:

- 0 The interface will not commit automatically.
- 4 A NOCOMMIT or COMMIT pipeline command has already been issued.
- 8 A read or write pipeline command has already committed the stage to level 0.
- 112 NOCOMMIT found operands.

Example: This example shows how to use NOCOMMIT and ADDPIPE to read a file on commit level -1. The return code is the number of lines in the file. This causes the remainder of the pipeline to be abandoned because the REXX program returns 11 without committing to level 0. Note that an equivalent CALLPIPE subroutine pipeline setting the variable directly commits the caller to level 0.

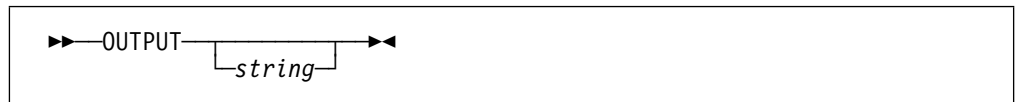
OUTPUT

TCMT REXX	Example of Use
<pre> /* TCMT REXX: Test nocommit */ signal on novalue trace error 'nocommit' 'addpipe (name TCMT)', '< tcmt rexx', 'count lines', '*.input:' 'readto line' exit line error: exit RC </pre>	<pre> pipe tcmt literal abc cons ▶R(00011); </pre>

Notes:

1. NOCOMMIT can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.

OUTPUT—Write a Line



Syntax: A string is optional with OUTPUT.

Operation: When issued from a REXX program, the stage commits to level 0 if the stage is not already committed unless NOCOMMIT has been issued to disable the implied commit operation. The argument string is written to the currently selected output stream. The record written begins after the blank ending the command verb; the record can have leading blanks. A null record is written when the string is omitted.

Return Codes:

- 0 The line is read by the stage connected to the output stream.
- 4 EOFREPORT ALL is in effect and a stream event has occurred before the consumer peeked at the record (or read it). The program should process the stream event and then reissue the OUTPUT command.
- 12 The output stream is not connected.
- 4095 The pipeline is stalled. All input streams and output streams are severed.
- ± The stage connected to the stream issued the pipeline command SETRC to set a return code.

Example:

HELLO REXX	Example of Use
<pre> /* HELLO REXX: REXX filter */ 'output' 'Hello, World!' </pre>	<pre> pipe hello console ▶Hello world! ▶Ready; </pre>

PEEKTO—Preview the next Input Line



Syntax: When present, the argument is the name of the variable that PEEKTO should set. The word must represent a valid name for a REXX variable as it would be written in a REXX program.

Operation: The stage commits to level 0 if the stage is not already committed unless NOCOMMIT has been issued to disable the implied commit operation. The next record on the currently selected input stream is copied into the variable. The record remains in the pipeline; use READTO to read or discard the line. Use PEEKTO without argument to test if the input stream is at end-of-file without setting a variable to the contents of the next record.

Return Codes:

- 0 The next record is available.
- 4 EOFREPORT ALL is in effect and a stream event occurred that did not cause return codes 8 or 12 to be set.
- 8 EOFREPORT ALL or EOFREPORT ANY is in effect. There is no longer a connected output stream.
- 12 The stream is at end-of-file. If a word is specified, the variable is dropped.
- 4095 The pipeline is stalled.

Example: PEEKTO is used after a subroutine pipeline to see if there are more input data to process.

HEADING REXX	Example of Use
<pre> /* A heading every 55 lines */ do until RC/=0 'callpipe (name HEADING)', *:', take 55', change // /', literal 1The heading', *:' 'peekto' end </pre>	<pre> pipe < heading rexx heading cons ▶1The heading ▶ /* A heading every 55 lines */ ▶ do until RC/=0 ▶ 'callpipe (name HEADING)', ▶ *:', ▶ take 55', ▶ change // /', ▶ literal 1The heading', ▶ *:' ▶ 'peekto' ▶ end ▶Ready; </pre>

The generic filter to pass records from the input to the output without delay and propagating end-of-file both forwards and backwards:

READTO

```
/* COPYND REXX -- Copy without potential to delay          */
Signal on novalue

'eofreport all'          /* Propagate EOF backwards too */
signal on error

do forever
  'peekto line'          /* Look for next input line    */
  /* Process line here */
  'output' line          /* Pass it to the output       */
  'readto'               /* Consume the record          */
end

error: exit RC*(wordpos(RC, '8 12')=0)
```

Notes:

1. PEEKTO can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.
2. If EOFREPORT ALL or EOFREPORT ANY is in effect, subsequent PEEKTO pipeline commands without intervening READTO pipeline commands will set return codes 4 or 8 when suitable stream events have occurred since the previous PEEKTO, even when an input record is available.

READTO—Read or Discard an Input Line



Syntax: When present, the argument is the name of the variable that READTO should set. The word must represent a valid name for a REXX variable as it would be written in a REXX program.

Operation: The stage commits to level 0 if the stage is not already committed unless NOCOMMIT has been issued to disable the implied commit operation. The next record on the currently selected input stream is copied into the variable and discarded.

No variable is set when the word is omitted; a record is discarded from the input stream if one is available.

Return Codes:

- 0 The next record is available.
- 12 The stream is at end-of-file. If specified, the variable is dropped.
- 4095 The pipeline is stalled.

Example: Use READTO to read records in a filter program.

COPY REXX	Example of Use
<pre>/* COPY REXX -- Copy unchanged */ signal on error Do forever 'readto record' 'output' record End error: exit RC*(RC<>12)</pre>	<pre>pipe literal a line copy cons ►a line ►Ready;</pre>

Notes:

1. READTO can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.

RESOLVE—Return Entry Point of Built-in Program

►►—RESOLVE—*word*—◄◄

Syntax: RESOLVE requires a word.

Operation: The word is looked up in the directories for built-in programs and attached filter packages. When positive, the return code is the entry point address.

Return Codes:

- 0 The name is not resolved as a built-in program or a program in an attached filter package.
- + The entry point address.
- 42 The argument is missing.
- 112 There is more than one word in the argument string.

Example: RESOLVE can test if a filter package is installed.

TRES REXX	Example of Use
<pre>/* Test RESOLVE */ 'resolve strings' If RC=0 Then 'message', 'TRESxx001E', 'Strings not installed'</pre>	<pre>pipe tres ►Strings not installed ►... Issued from stage 1 of pipeline 1 ►... Running "rexx tres"</pre>

REXXCMD—Call a REXX Pipeline Program from a Filter

►►—REXXcmd—*string*—◄◄

Syntax: The argument string to REXXCMD is the same format as for the *rexx* built-in program.

SCANRANGE

Operation: The program is called as a pipeline filter. It can access the caller's streams while it runs. The argument string is passed as the first argument string to the called program. The return code is the return code from running the program or the number of a message issued because the program could not be found.

Return Codes: The corresponding return code is set if message 21, 22, 40, 113, 122, 381, or 382 is issued. The return code is the return code from the program when the interface does not reflect an error.

Example: (PIPPCEND REXX is on the MAINT 193 disk; it generates an END card for an object module.)

TREXXC REXX	Example of Use
/* REXX pipeline command */ 'output Now follows the end card:' 'rexx pippcend' Exit RC	pipe trexxc console ►Now follows the end card: ► END ►Ready;

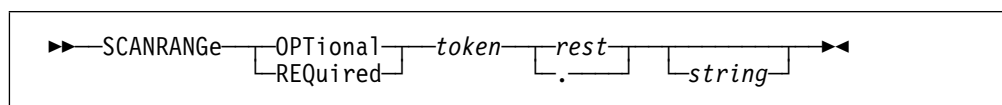
lhartma

Notes:

1. Using the pipeline command REXX in a REXX filter is equivalent to calling an external function in a command procedure. Variables are not shared between the caller and the called program.
2. The pipeline command environment is not available in an external function called from a REXX filter.
3. The function performed by REXX is also available with CALLPIPE where all connected streams are passed to a subroutine pipeline consisting of the one stage; REXX is retained for compatibility with the past.

SCANRANGE—Parse an input range

SCANRANGE can be used by the filter programmer to parse an argument string containing an *inputRange* specification in the same way that *CMS Pipelines* built-in programs scan their arguments when an *inputRange* can be specified.



Syntax: The first word is a required keyword:

OPTIONAL The argument string need not begin with a syntactically correct *inputRange*; the range 1-* is assumed instead.

REQUIRED The argument string must begin with a syntactically correct *inputRange*.

The second word specifies the name of a variable that will be set to a token representing the *inputRange*. The variable can be used in subsequent GETRANGE pipeline commands to select that input range. The contents of this token are unspecified; other tokens can represent other *inputRanges*. This variable must not be modified by the REXX program.

The third word specifies the name of the variable to receive the residual string after the *inputRange* specification has been scanned from the beginning of the argument string. Specify a period (.) to discard the residual string.

The remaining string is the argument string from which to scan the specification of an *inputRange*. Leading blanks are ignored.

: The default word and field separators are not remembered between instances of
 : SCANRANGE; the defaults apply to each invocation. Any qualifier set in the *inputRange*
 : specification is discarded when it has been parsed.

See also: GETRANGE.

Return Codes: Error messages have been issued when a nonzero return code is set. The program should deallocate any resources it may have allocated and exit with the return code.

Example: SCANRANGE is typically used at the beginning of a REXX filter where it scans its arguments string. For example, to scan an argument string that may specify one input range but no other arguments:

```
parse arg argstring
'scanrange optional range_definition rest' argstring
If RC~=0
  Then exit RC /* Messages are already issued */
If rest~=''
  Then call err 112, rest /* Too much */
```

To scan an argument string that must specify two input ranges and no other arguments:

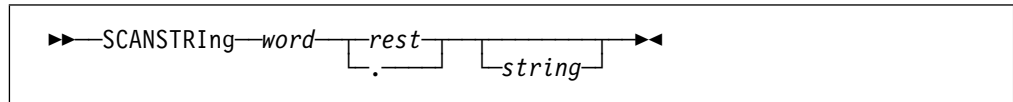
```
parse arg argstring
'scanrange required first rest' argstring
If RC~=0
  Then exit RC
'scanrange required second rest' rest
If RC~=0
  Then exit RC
If rest~=''
  Then call err 112, rest /* Too much */
```

Notes:

1. SCANRANGE can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.
2. The token representing the input range remains valid only as long as the stage is running. Once the stage has terminated, the contents of the token are stale. Using it (for example, in some other REXX filter) may cause random ABENDs.
3. The second and the third word are names of variables, but *string* represents a string.
4. To determine the length of the string scanned, subtract the length of the residual text from the length of the argument string.
5. PARSERANGE is a synonym for SCANRANGE.

SCANSTRING—Parse a delimited string

SCANSTRING can be used by the filter programmer to parse an argument string containing a *delimitedString* specification in the same way that *CMS Pipelines* built-in programs scan their arguments when a *delimitedString* can be specified.



Syntax: The first word specifies the name of a variable that will be set to the contents of the delimited string.

The second word specifies the name of the variable to receive the residual string after the *delimitedString* specification has been scanned from the beginning of the argument string. Specify a period (.) to discard the residual string.

The remaining string is the argument string from which to scan the specification of a *delimitedString*. Leading blanks are ignored.

Return Codes: Error messages have been issued when a nonzero return code is set. The program should deallocate any resources it may have allocated and exit with the return code.

Example:

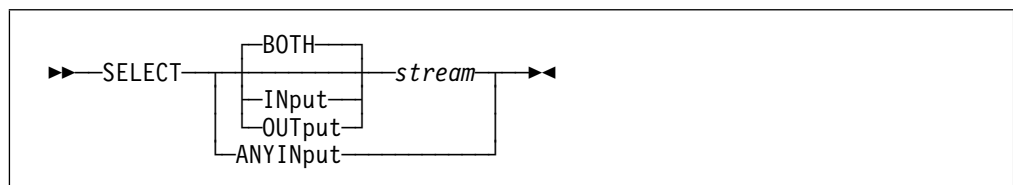
```
parse arg argstring
'scanstring word rest' argstring
```

The variable `argstring` could contain `/abc/` or `xf1f2f3`; the variable `word` would then be set to `abc` or `123`, respectively.

Notes:

1. SCANSTRING can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.
2. The first and the second word are names of variables, but *string* represents a string.
3. To determine the length of the string scanned, subtract the length of the residual text from the length of the argument string.
4. PARSESTRING is a synonym for SCANSTRING.

SELECT—Select a Stream



Syntax: The argument string to SELECT has an optional keyword followed by a stream identifier, or a keyword.

Operation: When the operation is SELECT ANYINPUT from a REXX program, the stage commits to level 0 if the stage is not already committed unless NOCOMMIT has been issued to disable the implied commit operation. The stream is selected on the side(s) specified. Use STREAMNUM to discover the number of the input stream selected when any input stream is desired.

Return Codes:

- 0 The stream(s) are selected.
- 4 When ANYINPUT is not specified, the stream does not exist; No message is issued.
- 4 EOFREPORT ALL is in effect and a stream event occurred that did not cause return codes 8 or 12 to be set.
- 8 EOFREPORT ALL or EOFREPORT ANY is in effect. There is no longer a connected output stream.
- 12 The pipeline command is SELECT ANYINPUT; all input streams are at end-of-file.
- 112 There are too many words in the argument string.
- 168 OUTPUT or BOTH is used with ANY.
- 169 Stream identifier not specified.

SETRC—Set Return Code in Stage Writing

►►—SETRC—*number*—◄◄

Syntax: SETRC requires a numeric argument.

Operation: In this description, the stage issuing SETRC is called the *consumer stage*. The stage connected to the currently selected input stream is called the *producer stage*.

It is verified that the stage connected to the currently selected input stream (the producer stage) is waiting for an output record to be read by the consumer stage on that particular stream. You can be sure of this only after a PEEKTO pipeline command and before the next READTO pipeline command.

The argument is stored as the return code that the producer stage sees when the consumer stage issues the next READTO pipeline command to consume the record.

Return Codes:

- 0 The return code is set
- 4 The currently selected input stream is not connected (the stage is first in a pipeline) or the stage connected to the currently selected input stream has selected a different output stream or it is not waiting for an OUTPUT pipeline command to complete. No message is issued.
- 58 The first word of the argument string is not a number.
- 112 There is more than one word in the argument string.
- 113 The argument string is empty.

Example: A REXX program to issue CP commands may feed the return code back using SETRC.

SEVER

```
/* Issue CP commands with RC */
'peekto in'
do while RC=0
  address command 'CP' in
  'setrc' RC
  'readto'
  'peekto in'
end
```

Notes:

1. SETRC should only be used when connected to programs that are prepared for any return code on OUTPUT.

SEVER—Break a Connection



Syntax: SEVER requires a keyword.

Operation: When the currently selected stream on the side specified is connected, these actions are performed at the stream at the other side of the connection: If the stream was created with CALLPIPE, the previous connection is reinstated. If the stream was not created with CALLPIPE, the stream becomes not connected; end-of-file is set if the stage is waiting for I/O on this stream or it is the last remaining input stream and the stage is waiting for any input stream.

For the stream specified at the stage issuing SEVER, the connection on the top of the ADDPIPE stack (if any) is reinstated. The stream becomes not connected if the stack is empty.

Return Codes:

- 0 The stream is severed.
- 112 There is more than one word in the argument string.
- 163 The argument string is empty.
- 164 The argument is not INPUT or OUTPUT.

Example: In a stage with more than one output stream, sever a stream as soon as you have finished writing to it. This may avoid a stall.

```
/* Process from label to secondary output */
parse arg label
'callpipe (name PIPCMDS)',
  '|*:',
  '|tolabel' label ||,
  '|*:'
'sever output'
'callpipe *:|procem|*.output.1:'
exit RC
```

Note: Though much *CMS Pipelines* documentation speaks of severing a stream rather than severing the connection to a stream, it is understood that the severance occurs by removing the connector between the stream being severed and the stream it is connected

to, if any. Streams are created by the pipeline specification parser and by the pipeline command `ADDSTREAM`; once created a stream exists as long as the stage to which it is attached. There is no pipeline command to destroy a stream.

SHORT—Connect Input and Output Stream

▶▶—SHORT—◀◀

Syntax: SHORT accepts no arguments.

Operation: The currently selected input stream and the currently selected output stream are connected directly, bypassing the stage issuing SHORT.

Return Code: 0.

Example: Use SHORT when you wish to copy all input to the output.

RDROP REXX	Example of Use
<pre>/* Drop records */ do word(arg(1) 1, 1) 'readto in' end 'short'</pre>	<pre>pipe < rdrop rexx rdrop 2 console ▶ 'readto in' ▶ end ▶ 'short' ▶ Ready;</pre>

STAGENUM—Return Stage's Position in Pipeline

▶▶—STAGENUM—◀◀

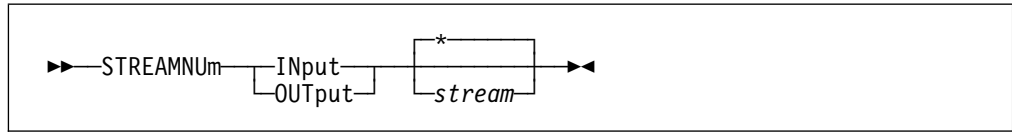
Syntax: STAGENUM accepts no arguments.

Operation: The return code is set to the position of the stage in the pipeline of its primary stream. The first stage gets return code 1.

Example: Use STAGENUM when you wish to ensure that a program is first (or not first) in a pipeline.

TPOS REXX	Example of Use
<pre>/* Test position */ 'stagenum' If RC=1 Then Do 'issuemsg 127 TPOSxx' exit RC End</pre>	<pre>pipe tpos ▶ This stage cannot be first in a pipeline ▶ ... Issued from stage 1 of pipeline 1 ▶ ... Running "rexx tpos" ▶ R(00127);</pre>

STREAMNUM—Return Stream Number



Syntax: STREAMNUM requires a keyword for the side to operate on. An asterisk, a number, or a stream identifier is optional.

Operation: The return code is set to the number of the stream. When the stream identifier is omitted or specified as an asterisk, the number associated with the currently selected stream is returned. When a number is specified as the stream identifier, it is verified that the stream exists; the return code is the number. The number of the stream that has the identifier specified is returned when the identifier is neither an asterisk nor a number.

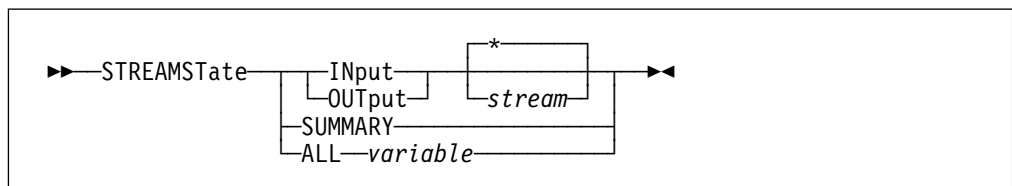
Return Codes:

- 0 The primary stream is selected or associated with the identifier.
- 4 The stream is not defined.
- 102 The second argument is a number. The stream does not exist.
- 112 There are more than two words in the argument string.
- 163 The argument string is empty.
- 164 The argument is not INPUT or OUTPUT.
- 178 The second argument is a stream identifier. The stream does not exist.

Example: Use STREAMNUM to test if there is a stream with a particular ID.

TSTRNO REXX	Example of Use
<pre>/* Test streamno */ parse arg stream_ID . trace off 'streamnum output' stream_ID If RC>=0 Then 'callpipe', ' literal Testing:', ' *..'stream_ID':'</pre>	<pre>pipe .dbg: tstrno gryf ▶Stream "gryf" not found ▶... Processing "streamnum output gryf" ▶... Issued from stage 1 of pipeline 1 ▶... Running "rexx tstrno gryf" ▶Ready; pipe .dbg: tstrno dbg console ▶Testing: ▶Ready;</pre>

STREAMSTATE—Return Stream Status



Syntax: STREAMSTATE queries the state of all streams or of a specified stream.

When SUMMARY is specified, the return code is zero if at least one input stream is connected and at least one output stream is connected. The program receives no indication of which streams are connected.

When ALL is specified, the state of each pair of input and output streams is stored as a word in the specified variable, which will contain as many words as the highest number of input or output streams. Each word contains the state of the input stream, a colon (:), and the state of the output stream. These states are defined in the "Return Codes" section below.

When neither SUMMARY nor ALL is specified, STREAMSTATE requires a keyword for the side to operate on; if it is present, the second word specifies the stream to test. The default is the currently selected stream.

Operation: The return code is set to indicate the summary status or the status of the specified stream.

The return code from STREAMSTATE ALL is zero unless there is trouble setting the variable.

Return Codes:

- 0 The stream is connected; the stage on the other side is waiting for I/O on this stream.
- 4 The stream is connected; the stage on the other side is waiting for I/O on this stream on a different commit level. The pipeline stalls if you try to read or write the stream before committing to the level the other side is at.
- 8 The stream is connected; the stage on the other side is not waiting for the stream.
- 12 The stream is not connected.
- 4 The stream is not defined.
- 112 There are more than two words in the argument string.
- 163 The argument string is empty.
- 164 The argument is not INPUT or OUTPUT.

Example: Use STREAMSTATE to test if an input or output stream is connected.

TSTRST REXX	Example of Use
<pre>/* Test streamstate */ trace off 'streamstate input 1' If find('12 -4', RC)>0 Then exit 0 'issuemsg 539 STRMST' exit RC</pre>	<pre>pipe tstrst ►Ready; pipe (end ?) t:tstrst?t: ►Ready; pipe (end ?) t:tstrst?hole t: ►Secondary input stream is connected ►... Issued from stage 1 of pipeline 1 ►... Running "tstrst" ►R(00539);</pre>

Notes:

1. STREAMSTATE ALL can be issued from a REXX program only; it is not available to *pipcmd* or the underlying PIPCMD macro.

SUSPEND

SUSPEND—Allow other Stages to Run

▶▶—SUSPEND—◀◀

Syntax: SUSPEND accepts no arguments.

Operation: The stage is put at the end of the dispatch list. The return code is set to the number of other stages that are ready to run at the time the stage was suspended. (That is, the return code is computed at the time the stage is suspended even though the stage must of necessity resume before it can inspect the return code.)

Return Codes:

- 0 There were no other stages ready to run. The stage was resumed immediately.
- + The number of stages that were ready to run at the time the stage was suspended.

Example: To try to obtain some confidence that an output record will be consumed:

```
'suspend'                                /* give consumer a chance to read */
'streamstate output'                     /* Now, did it? */
if RC<0                                  /* Not defined? */
  then exit RC                            /* This is an impossibility */
if RC=12                                  /* End-of-file; quit */
  then exit 0
'peekto line'                             /* Now try to read a line */
```

Notes:

1. The order of dispatching is unspecified. The pipeline dispatcher could select the suspended stage before it has run all stages that were ready at the time the stage issued SUSPEND. If the return code is positive, at least one other stage has run.
2. A program should not go into a loop waiting for a producer or a consumer to commit itself to write or read a record. Two stages using SUSPEND can be chasing each other's tails forever doing this.

Chapter 26. Message Reference

Messages issued by *CMS Pipelines* are listed in numerical order on the following pages. Use the CP command “set emsg on” to include the standard VM identification (module, number, and severity) in the message displayed on your terminal. Severity codes are defined in *z/VM: CMS and REXX/VM Messages and Codes*.

In addition to the reference in this chapter, you can obtain more information about a message in these ways:

- The command “pipe help” invokes help for the last message issued (excluding some informational messages). “pipe help 1” gives help on the second last error message, and so on for the last 11 messages.
- The messages are listed alphabetically in Appendix B, “Messages, Sorted by Text” on page 897.
- Unless disabled, additional messages (192 and 1 through 4) are issued automatically by *CMS Pipelines* to identify the stage or command causing the error.

In the list of messages on the following pages, the first line for a message is set in bold type. It gives the message number, severity code, and text. Words in the message text that are set in *italics* type are replaced with variable data.

Most stages return with the same return code as the number of the last message issued when the message indicates an error. Some errors are considered sufficiently grave to give negative return codes.

The text for a message displayed on your terminal is generated from the same Script input file that is used here. If you see a message not listed on the following pages or listed differently, then something is downlevel (though it could well be this book). To resolve this, type the command “pipe query” to obtain message 86 identifying the level of *CMS Pipelines* you are using.

0E No message text for message number

Explanation: *CMS Pipelines* has discovered an internal error. A *CMS Pipelines* module requests the message with the number shown, but there is no action defined for the message.

System Action: Depends on where the message is issued.

User Response: Ensure the message level is odd (it is unless you have changed it). Note the string substituted in message 1, if one follows. Contact your systems support staff.

System Programmer Response: If message 1 is issued and it indicates a REXX program, the program may have issued the *ISSUEMSG* pipeline command; ensure that it uses a correct message number. If message 1 is not issued, the unknown message is issued in the pipeline specification parser. Contact IBM for programming assistance if the pipeline module is unmodified and the number shown is not in the file *PIPELINE HELPIN*. If the message is defined in the *HELPIN* file, then ensure that the message text table is correctly generated and inspect the file *\$PIPE UPDLOG* to ensure that the correct version of it is included when generating *PIPELINE MODULE*.

1I ... Running "string"

Explanation: This message is issued after any other message when a stage is currently running and the message level is odd. The first 60 characters of the specification of the stage are substituted in the message.

System Action: None.

User Response: The message level is set by the command *PIPMOD MSGLEVEL*, by the global or local *MSGLEVEL*, and by the *MSGLEVEL* option on *runpipe*.

2I ... Processing "command"

Explanation: This message is issued after messages issued by the pipeline command processor if the bit for 2 is on in the message level. The first 60 characters of the pipeline command issued are substituted in the message.

System Action: Message 1 is issued if the message level is odd. Processing continues.

3I ... Issued from stage number of pipeline number

Explanation: This message is issued to identify which stage is the cause of the previously issued message when the option *NAME* is not used in the pipeline specification.

System Action: Message 1 is issued if the message level is odd. Processing continues.

4I ... Issued from stage number of pipeline number name "name"

Explanation: This message is issued to identify which stage is the cause of the previously issued message when the option *NAME* is used in the pipeline specification.

System Action: Message 1 is issued if the message level is odd. Processing continues.

10E Extended format parameter list is required

Explanation: *PIPE* is invoked with a call type flag, indicating that only a tokenised parameter list is available. Most likely you have entered the command from *EXEC1* or the command line of *BROWSE*.

Stages check that an extended parameter list is present and exit with return code -10 if the flag in the leftmost byte of register 1 indicates that no parameter list address is provided in register 0. No message is issued in this case because such an error indicates that the stage is not entered from the pipeline dispatcher.

System Action: The *PIPE* command or the stage returns with return code -10.

User Response: Use *CMDCALL* to issue a command using the required parameter lists when the environment does not build such parameter lists.

11E Null or blank parameter list found

Explanation: A null parameter list is found by *PIPE*, the pipeline command processor, or a stage needing parameters.

System Action: *PIPE*, the pipeline command processor, or the stage returns with return code -11.

12E Null pipeline

Explanation: The last character of a pipeline specification is the pipeline end character; two consecutive end characters are met; or global options are present (in parentheses) with no more data.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -12.

13E No ending right parenthesis for global options

Explanation: A leading left parenthesis is found, indicating global options, but there is no closing parenthesis.

System Action: Pipeline scan terminates with return code 13.

User Response: Terminate global options with a right parenthesis.

14E Option word not valid

Explanation: The word substituted is not recognised as one of the global options supported.

System Action: Pipeline scan terminates with return code 14.

User Response: Defined global options are: NAME TRACE LISTRC LISTERR LISTCMD STOP SEPARATOR ENDCHAR ESCAPE MSGLEVEL.

15E Value missing for keyword "keyword"

Explanation: An operand is specified that requires a value (for instance, NAME), but the following non-blank character is the right parenthesis that ends the global options, or the operand is the last word of the argument string to a stage. This message is issued when an option list ends prematurely, and by stages that use values with operands.

System Action: Pipeline scan terminates with return code 15. When issued by a stage, the stage returns with return code 15.

16E Last character is escape character

Explanation: The escape character (declared by the option ESCAPE) is the last character of a pipeline specification. This is an error because there is nothing to escape.

System Action: Pipeline scan terminates with return code -16.

17E Null stage found

Explanation: There is a stage separator at the end of a pipeline specification; a stage separator is adjacent to an end character; or there are two stage separators with only blank characters between them.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -17.

User Response: Ensure that the pipeline specification is complete. Check if a comma is missing to indicate REXX continuation to the next stage on the following line.

The INTM shell adds *console* stages to the beginning or end (or both) of a pipeline specification beginning or ending with a stage separator; the PIPE command does not.

Ensure that there are no blanks between what you intend to be a pair of self-escaping vertical bars (| |).

18E CMS Pipelines incorrectly generated with character

Explanation: *CMS Pipelines* has discovered an internal error. *CMS Pipelines* is generated with unacceptable characters for one of the delimiter characters (stage separator, left parenthesis, right parenthesis, period, or colon).

System Action: Pipeline scan terminates with return code -18.

User Response: Contact your systems support staff.

System Programmer Response: Restore a working copy of *CMS Pipelines*. Issue NUCXDROP PIPMOD followed by PIPINIT if the broken module was activated with "pipinit test". If not, you may have to resort to a backup of the product tape.

Correct the error introduced in the operand table (SYSTEM KWDTABLE by default) for the operands sc, lp, rp, cn, dt, and as.

19W Label "word" truncated to eight characters

Explanation: The first word of a stage ends in a colon; there are more than eight characters before the colon or the first period in the label.

System Action: The label is truncated on the right. Processing continues.

20I Stage returned with return code number

Explanation: Pipeline dispatcher trace is active; option LISTRC or option LISTERR is in effect. The stage has completed processing.

System Action: The pipeline dispatcher continues with other work. Control returns to the caller when all stages are complete.

21E Unable to find EXEC COMM for REXX

Explanation: *CMS Pipelines* has discovered an internal error. The EXEC interpreter did not set up a subcommand environment for EXEC COMM before issuing a command to the default command environment.

System Action: The REXX interface returns with code -21.

User Response: Contact your systems support staff.

System Programmer Response: Ensure that the pipeline module is generated correctly. This message indicates a change in the implementation of VM/REXX. Investigate whether corrective service is available.

23E Impossible record (number bytes from X'address')

Explanation: A stage writes a record (or tries to read into a buffer) that is completely or partially beyond the size of an address space for the virtual machine architecture (16M in a 370-mode virtual machine; 2G in an XA-mode virtual machine).

The contents of general registers zero and one are substituted.

System Action: Control returns to the stage with return code -23. The call is ignored.

User Response: Check the input file. If the contents of register zero are shown as negative, there may be a programming error in *CMS Pipelines*.

24W Descriptor list for program "command" is not doubleword aligned; it is ignored

Explanation: A PIPEXX macro is issued and the rightmost three bits of general register 2 are not all zero.

System Action: The program in storage is ignored.

User Response: Contact your systems support staff.

System Programmer Response: Make sure registers 2 and 3 are zero before issuing the macro PIPEXX to run a REXX program from disk as a pipeline stage.

26E Error number obtaining storage

Explanation: An error other than "no storage" is received from the system service to obtain storage. Note that the return code is displayed in hexadecimal.

System Action: The stage terminates with return code 26.

27E Entry point word not found

Explanation: The named entry point is not a built-in program; it is not found in any declared local directory; and there is no file with file name *word* and file type REXX.

System Action: Message 1 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -27. The stage terminates with return code -27 when this message is issued by a look up routine.

User Response: If the error seems to originate in a *specs* stage or some other stage that allows for a long and complex parameter list, check your pipeline specification to see if you have inadvertently used a stage separator or an end character without doubling it up to let it escape through the scanner. If you forgot, what you think of as a parameter list is, in fact, several stages or even several pipelines.

The usually useless piece of advice is: Verify the spelling of the name of the program to run. If, however, this shows that the name substituted is not the name you wrote, the pipeline specification has been truncated in the middle of the command verb. Most likely, message 1145 was also issued; address your CMS commands to COMMAND, not to XEDIT.

Issue "pipe query" to display the level of *CMS Pipelines* that you are using; compare the response with the response in this book. Contact your systems support staff if this book applies to the level of *CMS Pipelines* that you are using and the program is a built-in one.

System Programmer Response: If *CMS Pipelines* is installed in a shared segment, ensure that sufficient space has been allocated.

28I Starting stage with save area at X'address' on commit level number

Explanation: Pipeline dispatcher trace is active; the option STOP is specified. The address substituted designates the save area that contains the initial register set for the stage.

System Action: None.

29E Pipelines stalled

Explanation: A set of pipelines is deadlocked.

System Action: The state of each stage is listed in subsequent message 30s. PIPDUMP EXEC is called to write the pipeline control blocks to the file PIPDUMP LISTING A. All stages that have not completed have their input and output connections severed before being dispatched with return code -4095.

User Response: It may help to use *faninany* instead of *fanin*. Ensure that there is buffering in all loops when this does not help.

30I Stage is in state state

Explanation: The pipeline is stalled. The state of each stage is listed. The following states are defined:

ready	The stage is ready to run.
wait loc	Waiting for data in locate mode.
wait in	Waiting for data in move mode.
wait out	Waiting for a stage to read its output.
wait ecb	Waiting for an event control block to be posted.
unavail	The stage has been redefined by CALLPIPE; it waits for the subroutine pipeline to complete.
wait any	Waiting for data on any input stream.
returned	The stage has completed execution.
wait com	Waiting for other stages to commit.

System Action: None.

31I Resuming stage; return code is number

Explanation: Pipeline dispatcher trace is active. The stage is being resumed. The return code from the call to the pipeline dispatcher is shown.

System Action: None.

32I Storage address length

Explanation: When a pipeline specification is issued from *runpipe* TRACE, this message is issued before message 39 is issued to describe a data record and before message 34 is issued to indicate that a CALLPIPE or ADDPIPE pipeline command is being processed. The message text can be used as a pipeline stage to obtain the complete record or command.

33I Input requested for *number* bytes

Explanation: Pipeline dispatcher trace is active. A PIPINPUT macro or a READTO pipeline command is issued. The contents of register 0 are substituted for *number*.

System Action: None.

34I "entry point" called

Explanation: Pipeline dispatcher trace is active. The entry point shown is called.

System Action: Message 39 may follow with the pipeline command being issued.

35I Output *number* bytes

Explanation: Pipeline dispatcher trace is active. A PIPOUTP macro or a OUTPUT pipeline command is issued. The contents of register 0 are substituted for *number*.

System Action: Message 39 follows.

36I Select *side* stream *number*

Explanation: Pipeline dispatcher trace is active. A PIPSEL macro or a SELECT pipeline command is issued.

System Action: None.

37I Streamnum *side* stream number intersection *number*

Explanation: Pipeline dispatcher trace is active. A PIPSTRNO macro or a STREAMNUM pipeline command is issued.

System Action: None.

38I Setting dispatcher exit to X'*address*'

Explanation: Pipeline dispatcher trace is active. A PIPEXIT macro is issued.

System Action: None.

39I ... Data: "data"

Explanation: Pipeline dispatcher trace is active. The first 60 bytes of the record are shown.

System Action: None.

User Response: To see all data passing between two stages in the pipeline, insert a stage that copies the data to a file; then look at it later. Or write a REXX program to "say" the data.

When capturing a copy of the data flowing in the pipeline it may be important that end-of-file is propagated backwards through this device driver stage. Use *eofback* to run the device driver, for example:

```
... | eofback > trace file1 a | ...
```

40E REXX program *name* not found

Explanation: The *rexx* interface cannot find a file for the program you request. (Return code 8 on EXECSTAT.)

System Action: The stage terminates with return code 40.

41E Request "*code*" not valid on service call to *module*

Explanation: PIPMOD receives a service call with an argument that indicates neither that CMS ABEND processing is in process (PURGE) nor that the nucleus extension is being dropped (RESET). The first token of the argument is substituted.

System Action: The service call is ignored.

User Response: Contact your systems support staff.

System Programmer Response: Ensure that no program issues SVC 202 or CMSCALL with a call type of X'FF'.

42E Entry point missing

Explanation: The RESOLVE pipeline command is issued ! with no operands; *ldrtbls* or *nucext* has no operands; a ! required keyword is missing in the parameter list of a builtin ! program using a secondary entry point table.

System Action: Return code 42 or -42 is set.

43E Null label

Explanation: The first non-blank character of a stage definition is a colon.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -43.

User Response: Use any one of these ways to invoke a REXX program that has a colon in its name:

- Add *rexx* to specify that the program is in REXX.
- Define an escape character; put the escape character before the colon.
- Add a null label (:).

44E Label *string* is not valid

Explanation: *string* does not conform to the syntax for a label. For instance, there may be two or more periods in it.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -44.

45W Stream identifier "*name*" truncated to four characters

Explanation: There are more than four characters between the period beginning a stream identifier and the colon ending the label.

System Action: The stream identifier is truncated. Processing continues.

46E Label *label* not declared

Explanation: No specification for a stage is found the first time the label is used. The first usage of a label defines the stage to run, and any operands it may have. Subsequent references are to the label by itself.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -46.

User Response: Ensure that the label is spelt correctly. If this is the case, inspect the pipeline specification to see if a stage separator is erroneously put between the label and the verb for the stage.

47E Label *label* is already declared

Explanation: A reference is made to a label that is already defined. The label reference should be followed by a stage separator or an end character to indicate reference rather than definition.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -47.

User Response: Ensure that the label is spelt correctly. If this is the case, add a stage separator after the label to indicate that this is a reference to a stream other than the primary one. Note that all references to a label refer to the invocation of the stage that is defined with the first usage of the label.

48E Conflicting value for keyword *keyword*: *character*

Explanation: The character used for the operand is already a special character in the pipeline specification parser.

The following characters are reserved for other uses in a pipeline specification: left and right parentheses “()”, colon “:”, period “.”, blank, asterisk “*”. Other special characters are defined by the global options option SEPARATOR (by default “[”]), option ENDCHAR, and option ESCAPE.

System Action: Scan terminates with return code 48.

User Response: Use another character for the function.

49E Value for keyword "*keyword*" is not acceptable

Explanation: The value must be a single character or two-digit hexadecimal representation of the character to be used for the function indicated by the operand.

System Action: Scan terminates with return code 49.

**50E Not a character or hexadecimal representation:
*word***

Explanation: *word* is not a character or a two-digit hexadecimal representation of a character.

System Action: Return code 50 is set. If issued from the scanner, scan terminates with return code 50. If issued from a stage, the stage terminates with return code 50.

51E Missing operand after *inputRange(s)*

Explanation: A column range or a list enclosed in parentheses is specified, but no further operands are present.

System Action: The stage terminates with return code 51.

User Response: Specify the range “*-*” if you wish to change left parentheses; or rearrange the list of translations so that the first one is not the beginning of a valid range.

52E Unknown translate table "*word*"

Explanation: The table is not INPUT, OUTPUT, LOWER, UPPER, A2E, or E2A; nor is it one of the operands TO or FROM (which designate a codepage number).

word is the first word specified after column ranges, if any. It is neither a translation specification nor one of the operands designating a translate table.

System Action: The stage terminates with return code 52.

53E Odd number of translate pairs

Explanation: The argument string ends prematurely.

System Action: The stage terminates with return code 53.

User Response: The most likely cause of this error is that the first operand is interpreted as a column range instead of a translation specification. For instance, “xlate 40 a” gives this message instead of translating blank characters to lower

case "a". "xlate space a" or "xlate 1-* 40 a" performs the intended function.

54E Range "numbers" not valid

Explanation: A list of column ranges is opened or a keyword is specified that indicates a list of words or fields. A word in it does not conform to a column range syntax. If a valid decimal range is specified, the beginning column is zero or the end of the range is before the beginning.

System Action: The stage terminates with return code 54.

User Response: Add the range "*_*" if you wish to translate the left parenthesis; or rearrange the list of translations so that the first one is not the beginning of a valid range.

55E No inputRange(s) in list

Explanation: A left parenthesis is found, which indicates the beginning of a list of input ranges. The next non-blank character is a right parenthesis, which indicates that the list contains no ranges.

System Action: The stage terminates with return code 55.

User Response: Add the range "*_*" for the column range if you intend to translate left parentheses to right parentheses, like this: "xlate *-* ()"; or rearrange the list of translations so that the first one is not the beginning of a valid range.

56E More than 10 inputRanges specified

Explanation: There are more than 10 words in the list of column ranges.

System Action: The stage terminates with return code 56.

User Response: Use a cascade of *xlate* if you need to translate more than 10 ranges. Alternatively, use a subroutine pipeline with a *spec* to put the fields to be translated adjacent to each other; perform the transliteration desired; then use another *spec* to put them back where they were in the input record.

57E Missing right parenthesis after inputRanges

Explanation: A left parenthesis is found, meaning a range of columns is specified, but no closing right parenthesis is found.

System Action: The stage terminates with return code 57.

User Response: Add the range *-* if you wish to translate left parentheses; or rearrange the list of translations so that the first one is not the beginning of a valid range.

58E Decimal number expected, but "word" was found

Explanation: The word contains a character that is not a digit.

System Action: The stage terminates with return code 58.

59E Logical record length *number* is not valid

Explanation: The number is zero or negative.

System Action: The stage terminates with return code 59.

60E Delimiter missing after string "string"

Explanation: No closing delimiter is found for a delimited string.

System Action: The stage terminates with return code 60.

User Response: Most likely you never intended to specify a delimited string, but a mistake in a column range caused the specification error.

61E Output specification missing

Explanation: The output column is not specified for the last item.

System Action: The stage terminates with return code 61.

User Response: A likely cause is that an earlier specification is interpreted as a delimited string instead of what it was intended to be.

62E Command length *number* too long for CP

Explanation: The argument string or an input line is longer than the 240 bytes supported by CP, even after leading blank characters are stripped.

System Action: The stage terminates with return code 62.

63E Output specification *word* is not valid

Explanation: The word specifies where to put a field in the output record; it is not a positive number or a column range.

System Action: The stage terminates with return code 63.

User Response: A mistake in a conversion or placement option can trigger this message. Another likely cause is that an earlier input specification has been scanned as a delimited string where it should have been a column.

64E Hexadecimal data missing after *prefix*

Explanation: A prefix is found, indicating that a hexadecimal constant should follow, but the next character is blank or the end of the argument string.

System Action: The stage terminates with return code 64.

User Response: Do not use letters as delimiters for a delimited string.

65E "string" is not hexadecimal

Explanation: An h, H, x, or X is found in the first character of a specification item to specify a hexadecimal literal, but the remainder of the word is not composed of hexadecimal digits.

System Action: The stage terminates with return code 65.

User Response: Do not use letters as delimiters for a delimited string.

66E Number *number* is outside the valid range

Explanation: The number is not appropriate in the context where it is used.

System Action: The stage terminates with return code 66.

67E The number is incompatible with "option"

Explanation: A number is found first in the argument string, indicating a modifier, but the AT option implies that the target is removed when splitting. *split* cannot split before or after this target.

System Action: The stage terminates with return code 67.

68E Incorrect OS block descriptor word X'hex'

Explanation: The first four bytes of an input record are substituted. The last two bytes of the block descriptor word are not zero.

System Action: The stage terminates with return code 68.

User Response: Check the input file. The most likely cause is that the file you are trying to deblock is not a variable format OS file with block and record descriptor words.

69E Block size mismatch; *number* bytes read, but block descriptor word contains *number*

Explanation: The block size in the block descriptor word does not agree with the amount of data read.

System Action: The stage terminates with return code 69.

User Response: Check the input file. The most likely cause is that the file you are trying to deblock is not a variable format OS file with block and record descriptor words.

Another reason for this error may be that the file has been edited and trailing blank characters were removed when it was stored by XEDIT.

Spurious characters have been observed at the end of monitor records.

The following REXX program pads or truncates a record to the length indicated in the block descriptor word.

```
/* FIXBDW REXX -- make as long as BDW says */
signal on error
do forever
  'readto in'
  'output' left(in,c2d(left(in,2)))
end
error: exit RC*(RC<>12)
```

70E Incorrect OS record descriptor word X'hex'

Explanation: The last byte of the record descriptor word is not zero. If less than 4 bytes of hexadecimal data are substituted, there may be spurious data at the end of a block.

System Action: The stage terminates with return code 70.

User Response: Check the input file. The most likely cause is that the file you are trying to deblock is not a variable format OS file with block and record descriptor words.

71E Column number "number" must be positive

Explanation: The number is not positive.

System Action: The stage terminates with return code 71.

72E Last record not complete

Explanation: Premature end-of-file is received by *deblock* CMS or *unpack*. That is, end-of-file is received in the middle of a logical record.

System Action: The stage terminates with return code 72.

User Response: Check the input file. The most likely cause is that the input is not blocked with CMS record descriptor words or not in packed format.

73E Segmentation flags not compatible; previous is X'*previous*' and current is X'*current*'

Explanation: The segmentation flags are incompatible in the sense that the end of a record is not followed by the beginning of another one, or a segment that is not the end of a record is followed by a segment indicating the beginning of a record.

System Action: The stage terminates with return code 73.

User Response: Check the input file. The most probable cause of this error is that the data set is not in the specified format. When using *deblock* NETDATA on a reader file, be sure to:

- Select only records with X'41' in the first column.
- Delete the first column.
- Pad the record to 80 bytes.

74E Fixed records not same length; last bytes followed by current bytes

Explanation: Input records to *block* FIXED are not all the same length.

System Action: The stage terminates with return code 74.

User Response: Check the input file. Maybe you wanted the function performed by *fblock* rather than *block*; *fblock* accepts records of any length. Use *pad* to increase the length of short records, *chop* to truncate records.

75E Block size not integral multiple of record length; remainder is number

Explanation: The block size specified is not an integral multiple of the length of the first record read. For *fbawrite* : the block less the prefix does not contain a multiple of 512 : bytes.

System Action: The stage terminates with return code 75.

User Response: Use *fblock* if you wish to combine records irrespective of their lengths.

76I Waiting on ECB at X'address': hex

Explanation: Pipeline dispatcher trace is active. The stage issues the macro PIPWECB. The address of the ECB and its contents are shown. Bit 1 of the ECB (X'40') indicates that it is posted.

System Action: Processing continues.

77I Return code number

Explanation: The return code from a pipeline command is not zero. The option LISTERR is active.

System Action: None.

78E Record length number is too much

Explanation: The input record is too long for the device driver or blocking filter in question.

System Action: The stage terminates with return code 78.

User Response: Check the input file. *block* CMS, *disk*, *fullscr*, *printmc*, *punch*, and *uro* only accept up to 65535 bytes of data.

block V and *block* VB do not support input records longer than 32752 bytes (which is equivalent to the OS restriction of 32756 including the record descriptor word). Use *block* VBS to process records of any length.

79E CCW command code X'hex' is not valid

Explanation: Except for X'5A', the first byte of a record does not contain a write or control CCW: the rightmost bit is zero.

System Action: The stage terminates with return code 79.

User Response: Check the input file. Most likely there is no CCW operation code in the first column of the data record. Use *punch* instead of *uro* or *printmc* if you do not need to create records that have no operation carriage control.

80E More than 255 conversion triplets specified

Explanation: More than 765 operand words are found; *overstr* cannot handle more than 255 triplets.

System Action: The stage terminates with return code 80.

User Response: Build a cascade of *c14to38* filters if you need to process that many combinations.

81E Incomplete conversion triplet

Explanation: The number of operand words is not divisible by three.

System Action: The stage terminates with return code 81.

82E Device address word is not hexadecimal

Explanation: The device address shown is not composed of hexadecimal characters.

System Action: The stage terminates with return code 82.

User Response: Ensure operands are spelt correctly; a misspelled keyword is interpreted as a device address.

83E Device word does not exist

Explanation: CP sets condition code 3 on diagnose 24, indicating that the virtual device does not exist.

System Action: The stage terminates with return code 83.

User Response: Ensure operands are spelt correctly; a misspelled keyword is interpreted as a device address.

84E Virtual device word is not a supported virtual type

Explanation: The virtual device class and type returned for the device are not compatible with the function requested. For instance, it is not a printer for *printmc* or not a punch for *punch*.

System Action: The stage terminates with return code 84.

85E Virtual device word is not a supported real type

Explanation: The real device class and type returned for the device are not compatible with the function requested.

System Action: The stage terminates with return code 85.

86I CMS Pipelines, 5741-A07 modlevel
(Version.Release/Mod) - Generated April 29, 2020
at 2:50 p.m.

Explanation: This is the response to the command PIPE QUERY. Date and time shown here represent the time when this book was formatted. In the actual message they are replaced with the date and time the module was generated. This time is normally less than a minute before the timestamp of PIPELINE MODULE, unless the file has been transported with SENDFILE across time zone boundaries or between systems with dissimilar time zone specification in DMKSYS or HCPSYS.

System Action: Return code 86 is set.

87E This stage must be the first stage of a pipeline

Explanation: A program that cannot process input records is not in the first position of the pipeline.

System Action: The stage terminates with return code 87.

88E Buffer overflow

Explanation: *buildscr* needs more than 16K to build the screen image.

System Action: The stage terminates with return code 88.

User Response: Check the input file. Ensure (for instance with *asatomic*) that the input file does have machine carriage control.

89E Return code number reading the virtual reader

Explanation: Diagnose 14 sets the return code shown.

System Action: The stage terminates with return code 89.

User Response: One reason is that the first file in the reader is a VMDUMP file and you did not use the option 4KBLOCK. Refer to *z/VM CP Programming Services*, SC24-6272, for a description of the error codes for diagnose 14.

90E No reader file available

Explanation: There are no files in your reader that can be processed.

System Action: The stage terminates with return code 90.

User Response: The option MONITOR, or lack thereof, determines which type of file *reader* tries to read. There may still be files ready for reading of the other kind (monitor or not, as appropriate).

91E Return code number from CONSOLE type macro

Explanation: The CONSOLE interface is used to a 3270 terminal and the return code shown is received from CMS. *type* displays the second doubleword of the CONSOLE parameter list.

System Action: The stage terminates with return code 91.

User Response: The return codes from CONSOLE are described in the *z/VM CMS Macros and Functions Reference*, SC24-6262.

92E More than ten key fields

Explanation: More than the maximum ten key fields are specified for *sort* or *merge*.

System Action: The stage terminates with return code 92.

User Response: Use *spec* to rearrange the records to make the fields contiguous so that they can be coalesced.

93E Pipeline not installed as a nucleus extension; use PIPE command

Explanation: *CMS Pipelines* is initialised, but general register 2 does not point to an SCBLOCK for its entry point. This message has also been observed when a downlevel or modified NUCXLOAD MODULE is used to install the pipeline module as a nucleus extension.

System Action: Processing terminates with return code 93.

User Response: Use the command PIPE to run a pipeline specification. Do not issue the command PIPELINE or NXPIPE; these modules run in the user area when invoked directly as a command.

System Programmer Response: Ensure that the PIPELINE MODULE is relocatable.

94E Token token is not valid for PIPMOD

Explanation: The PIPMOD command is issued with flag byte zero. The subcommand is not supported.

System Action: Processing terminates with return code 94.

User Response: Do not issue the PIPMOD command from an EXEC1.

95E Operand word is not valid for PIPMOD

Explanation: The PIPMOD command is issued from the command line or from an EXEC. The word shown is not supported.

System Action: Processing terminates with return code 95.

96E Missing PIPMOD operand

Explanation: The PIPMOD command is issued with a register 1 flag byte of zeros. No arguments are found.

System Action: Processing terminates with return code 96.

97E Userword for pipe nucleus extension is zero

Explanation: CMS Pipelines has discovered an internal error. The nucleus extension for PIPE is installed, but no pipeline header is allocated.

System Action: Processing terminates with return code 97.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in CMS Pipelines.

98E Connector not by itself

Explanation: A label is found that has an asterisk as the first component, but a stage definition follows.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -98.

User Response: Connectors must be at the beginning or the end of a pipeline. Most likely an end character or a stage separator is missing.

99E Connector not at the beginning or the end of a pipeline

Explanation: A connector is in the middle of a pipeline.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -99.

User Response: Connectors specify how to couple streams between the active pipeline and the one being added to the pipeline set. Connectors must be at the beginning or the end of a pipeline. Most likely an end character or a stage separator is missing.

100E Direction "word" not input or output

Explanation: The second component of a connector is not a recognised operand.

System Action: Pipeline scan terminates with return code 100.

101E Connector connector can be specified with ADDPIPE or CALLPIPE

Explanation: A connector is found, but the pipeline was not issued with an ADDPIPE or CALLPIPE.

System Action: Pipeline scan terminates with return code 101.

102E Stream number not defined

Explanation: A connector requests the stream with the number shown, but the calling stage does not have that many streams defined.

System Action: Pipeline scan terminates with return code 102.

User Response: Note that the primary stream has number 0; the secondary stream is number 1.

103E Stream identifier not defined

Explanation: The calling stage does not have a stream with the identifier specified in the third component of the connector.

System Action: Pipeline scan terminates with return code 103.

104E Compiler stack overflow

Explanation: CMS Pipelines has discovered an internal error. A filter is out of space for its compiler stack while generating code.

System Action: The stage terminates with return code 104.

User Response: Try to reduce the complexity of the argument string. Contact your systems support staff.

System Programmer Response: This is a programming error in CMS Pipelines. Investigate whether corrective service is available.

105E Compiler overflow

Explanation: A filter is compiling a program; the program is too large to fit into the area that has been allocated for this purpose.

System Action: The stage terminates with return code 105.

User Response: For *sort*, the fields required such a complex program to compare that the available storage is exhausted. Use *spec* to rearrange the sort fields to become less complex. *sort* has read all input, but produces no output.

System Programmer Response: This is **not** a programming error in CMS Pipelines.

107E PIPMOD nucleus extension dropped before PIPE command is complete

Explanation: The PIPMOD nucleus extension is dropped while a pipeline is active.

System Action: Control is returned to CMS with return code 107. Results are unpredictable when control returns to the pipeline; an ABEND is likely.

User Response: Do not issue PIPINIT or NUCXDROP PIPMOD while a pipeline specification is being run.

108E Return code number from operation operation on tape tape

Explanation: CMS refuses to perform an I/O operation on the tape drive.

System Action: The stage terminates with return code 108.

User Response: Refer to the RDTAPE or WRTAPE macro description in the *z/VM CMS Macros and Functions Reference*, SC24-6262, for a description of the error codes.

Error 2 on write means that end of tape is reached while writing tape mark(s) after the file; all input records have been processed.

109E Keyword word is not a valid blocking format

Explanation: The operand is not Fixed, Variable, C, and so on.

System Action: The stage terminates with return code 109.

User Response: *deblock* v supports all OS variable record formats, blocked or spanned, or both.

110E Unsupported record in IEBCOPY unloaded data

Explanation: The top three bits of the first record are not all zero.

System Action: The stage terminates with return code 110.

User Response: Check the input file. If the data is indeed an IEBCOPY unloaded PDS, then there seems to be a note list. Remove it in a *drop* or *nfind* stage.

111E Operand word is not valid

Explanation: A keyword operand is expected, but the word does not match any keyword that is valid in the context.

System Action: The stage terminates with return code 111.

112E Excessive options "string"

Explanation: A stage has scanned all options it recognises; the string shown remains.

System Action: The stage terminates with return code 112.

User Response: This error may occur when a delimited string is intended, but a single character is found. For example, in "chop / ,/|" the first forward slash means the

literal character '/' rather than the opening of a delimited string. Though "chop /, /|" would scan the intended way, the preferred specification is "chop any / ,/".

Another likely cause is that a stage separator is missing and what is intended as a following stage is treated as additional operands to the current stage.

113E Required operand missing

Explanation: A stage has found some, but not all, required operands.

System Action: The stage terminates with return code 113.

114E Block size missing

Explanation: *block* is issued without an operand.

System Action: The stage terminates with return code 114.

User Response: Specify the block size for a default of Fixed.

115E Block size too small; number is minimum for this type

Explanation: The block size is too small to hold a record or segment, even of one byte.

System Action: The stage terminates with return code 115.

116E File type missing

Explanation: The argument string is one word (the file name).

System Action: The stage terminates with return code 116.

User Response: Write file names, types, and modes as blank-delimited words. Specify both the file name and the file type.

117E File mode "word" longer than two characters

Explanation: Three or more characters are found in the third word of the argument string.

System Action: The stage terminates with return code 117.

118E Return code number from renaming the file

Explanation: An erase and write operation is requested for a file. The file exists, so a utility file is written and renamed. The RENAME function fails with the return code shown.

System Action: The stage terminates with return code 118.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *CMS Pipelines*. Investigate whether corrective service is available.

119E Mode letter not available or read only

Explanation: A mode letter is specified; the mode letter is not accessed, or it is accessed read only.

System Action: The stage terminates with return code 119.

User Response: Most likely mode A is not accessed when a file is to be written there. Another cause could be that the one character abbreviation of the record format is written without a file mode; add a mode letter or directory. Yet another cause could be that you are accessing an SFS directory that belongs to some other user; CMS will access this directory read only, even if you have write privileges, unless you specify the option FORCERW on the ACCESS command.

120E Return code error number from parameter list function fn ft fm

Explanation: An unexpected return code is returned from the CMS file system. *function* displays the first token of the parameter list. It is 'Vblockw' when the full block interface is used to create a file. The last three words show the name, type, and mode of the file.

System Action: The stage terminates with return code 120.

User Response: Error code 3 often means that a file has been replaced on a shared minidisk. Access the disk again if this is the case. Consider moving the file to an SFS directory.

Refer to the error codes for the FSREAD and FSWRITE macros in the *z/VM CMS Macros and Functions Reference*, SC24-6262.

Error code 15 when function is 'Vblockw' can mean that CMS has updated the disk file directory while a file was being created. This is normally caused by the ERASE command. To circumvent this, use *diskslow* or buffer the file in *buffer* before writing it with *disk*. Write the file to an SFS directory or a separate minidisk if the file is too large to buffer and *diskslow* performance is not acceptable.

! Error code 16 when function is 'Vblockw' has been observed when two > stages were writing the same file.

121E File not found in the active file table

Explanation: *CMS Pipelines* has discovered an internal error. Having written a file through the full block interface, the *disk* device driver is unable to find the AFT entry for the newly created file.

System Action: The stage terminates with return code 121.

User Response: The reason may be that the file has been closed during execution, possibly by some other stage going

into subset. Use *diskslow* to overcome this problem or buffer the file with *buffer* before writing it.

122E Insufficient free storage

Explanation: A stage requesting storage has received a nonzero return code.

System Action: The stage terminates with return code -122. There may be too little storage left even to issue this message. In that case, the message is suppressed, but the pipeline return code is likely to be -122.

123E Not same ADT

Explanation: *CMS Pipelines* has discovered an internal error. *disk* found an entry in the active file table describing the file being written, but it seems not to be on the disk it should be.

System Action: The stage terminates with return code 123.

User Response: Contact your systems support staff. Use *diskslow* to write the file.

System Programmer Response: The error is either in the *disk* device driver or in DMSLAF, or the format of the parameter list to DMSLAF has changed.

This message is issued by modification levels 0 through 2 of *CMS Pipelines* when *disk* is used to write variable format files on VM/System Product Release 6. The file is written correctly. Install the current modification level.

This message has also been observed when the program level in NUCON was changed to indicate a release earlier than 6 when the system was in fact release 6 or later.

124E Error reading file: Length of record is number but file has logical record length number

Explanation: A V format file is being read through the full block interface. A record is met with a length field indicating a length longer than the logical record length in the file status table entry for the file.

This error can occur when a file on a shared minidisk has been updated by another virtual machine after you have accessed the minidisk.

System Action: The stage terminates with return code 124.

User Response: Access the disk and try again if the file is on a shared minidisk. Contact your systems support staff if you can XEDIT the file or read it with *diskslow*.

System Programmer Response: This may be a real error in the file system, or it may be a programming error in the *disk* device driver. You can disk dump a file to yourself and read it with READCARD if you wish to see the record descriptor words.

125E File mode missing

Explanation: An erase and write operation (>) is requested for a file without specifying the file mode.

System Action: The stage terminates with return code 125.

User Response: Specify the mode letter where you wish to write the file.

126E File mode * not allowed

Explanation: An erase and write operation (>) is requested with an asterisk as the file mode.

System Action: The stage terminates with return code 126.

User Response: Specify the mode letter where you wish to write the file.

127E This stage cannot be first in a pipeline

Explanation: A device driver that requires an input stream is first in a pipeline, where there can be no input to read.

System Action: The stage terminates with return code 127.

128E Record format not existing file format *letter*

Explanation: A file is to be appended to. The explicit record format specified is not the same as the one for the existing file.

System Action: The stage terminates with return code 128.

User Response: Specify the correct record format; use > to replace a file; or erase the existing file before issuing the pipeline.

129E Error reading file: Premature end of file

Explanation: A V format file is being read through the full block interface. The end-of-file record is not expected.

This error can occur when a file on a shared minidisk has been updated by another virtual machine after you accessed the minidisk.

System Action: The stage terminates with return code 129.

User Response: Access the disk and try again if the file is on a shared minidisk. If the file is indeed in order, there is a programming error in *disk*. Use *diskslow* instead.

131E Specified logical record length does not match existing logical record length *number*

Explanation: A file that has fixed record format is to be appended to. The record length specified is not the same as the one for an existing file.

System Action: The stage terminates with return code 131.

User Response: Specify the correct record length or erase the existing file.

132E Stream "word" already replaced

Explanation: The stream is requested to be replaced in two or more connectors. For instance, two or more connectors refer to **.input:* at the beginning of a pipeline.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -132.

133E Stream "word" already prefixed

Explanation: The stream is referenced in two or more connectors that specify a prefix type connection. For instance, two or more connectors refer to **.input:* at the end of a pipeline.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -133.

134E Record is *number* bytes, but format F file record length is *number*

Explanation: While file that has fixed record format is being written, an input record does not have the correct length.

System Action: The stage terminates with return code 134.

User Response: Check the input file. Use *pad* to extend records; *chop* to truncate.

137E The string of operands is too long

Explanation: The operand string for *asmfind* or *asmnfind* is longer than 71 bytes, indicating that the target is not entirely in the first record. The argument string to *sql* is 32K or longer. An input line to *attach* is 32K or longer.

System Action: The stage terminates with return code 137.

User Response: Use *asmcont* to combine continuation records before *find* or *nfind*; reconstruct the Assemble file with *asmxpnd*.

138E Short circuit not from input to output in *connector*

Explanation: Two connectors are in a pipeline of their own with no stage between them. In this case, the first one must be for input and the second one must be for output.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -138.

139E No connection available to redefine for *connector*

Explanation: A redefine operation is attempted with a ADDPIPE or CALLPIPE pipeline command, but the connection is severed, and thus there is no connection to redefine.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -139.

140E Record longer than specified length *bytes bytes*

Explanation: An input record does not fit in the buffer when creating format V or VB records or a record is longer than an explicit length on *pack VARIABLE*. The length of the record is substituted.

System Action: The stage terminates with return code 140.

User Response: Check the input file. Increase the block size to accommodate the required length if you are indeed blocking the data you intend to block.

141E XEDIT not active

Explanation: The *xedit* device driver is invoked; it finds no active XEDIT subcommand environment. (The return code from CMS is -3.)

System Action: The stage terminates with return code 141.

142E File "*fn ft fm*" is not in the XEDIT ring

Explanation: The named file is not in the active XEDIT ring; the return code from XEDIT is 28.

System Action: The stage terminates with return code 142.

User Response: Use the XEDIT subcommand of XEDIT to load a file into the ring.

143E Return code *number* from XEDIT state

Explanation: A return code not -3 or 28 is received on STATE of a file in XEDIT. Return code 24 means that the file name or the file type begins with X'FF'.

System Action: The stage terminates with return code 143.

User Response: Ensure the file is correctly named. If so refer to the description of the return codes in "Using XEDIT to Access Files in Storage" in *z/VM CMS Application Development Guide for Assembler*, SC24-6257.

144E Return code *number* from XEDIT operation

Explanation: The return code shown is received from XEDIT. Error 13 has been observed when there is no more storage to insert lines.

This message is also issued when it is not possible to find the subcommand environment to transport data; message 145 indicates the function as SUBCOM.

System Action: The stage terminates with return code 144.

User Response: Try the error codes listed for FSREAD and FSWRITE in *z/VM CMS Macros and Functions Reference*, SC24-6262.

145I Requesting *function* on *fn ft fm*

Explanation: The last three words show the file being accessed. The function is DMSXFLST for a STATE request; it is DMSXFLRD to read a line from the file; it is DMSXFLWR to write a line into the file.

System Action: None.

146E File "*fn ft fm*" does not exist

Explanation: A file does not exist. It is requested by *pdsdirect*, *members*, or file read is requested with the synonym <.

System Action: The stage terminates with return code 146.

User Response: Use *disk* to treat missing files as if they have no records.

147E File not a proper PDS

Explanation: The first record of the file does not contain a recognised identifier.

System Action: The stage terminates with return code 147.

148E Directory pointer *number* not compatible with file of size *number*

Explanation: The record number of the directory for the simulated partitioned data set is less than two or larger than the number of records in the file.

System Action: The stage terminates with return code 148.

User Response: Ensure that the data set is generated correctly. This error is reported when reading a *maclib* that is not generated completely. A null directory is identified while the library is being built; a failure in *maclib* (for instance disk full) can leave the output file in a state that is not valid.

149E Offset is not smaller than modulo

System Action:

Explanation: The first number specified must be zero or positive and smaller than the second. The stage terminates with return code 149.

150E • 167E

150E Member *word* not found

Explanation: The member listed does not exist in the library.

System Action: The stage terminates with return code 150.

User Response: When extracting members from a TXTLIB, *members* requires the name of the first CSECT in an object module. It does not resolve entry points the way the CMS loader does.

151E Operand "*string*" is not range of characters or a delimited string

Explanation: The operand is neither a range of characters nor a *delimitedString* of enumerated characters.

System Action: The stage terminates with return code 151.

152E Block size *number* too large; *number* is the maximum

Explanation: The block size for *block* is larger than the size supported for the blocking format in question. For V and the three other variable formats, the maximum is 32760. For AWSTAPE, the maximum is 65541.

User Response: Choose a smaller block size.

154E Operating environment not supported by stage

Explanation: A stage is requested which does not run on the operating system at hand.

System Action: The stage terminates with return code 154.

155E "*attribute*" is not three characters or hexadecimal

Explanation: One of the first four words in the arguments to *buildscr* is neither an asterisk nor three characters.

System Action: The stage terminates with return code 155.

User Response: Write three characters for extended attributes, or a single asterisk.

156E String missing

Explanation: An operand (for instance, ANYOF) is found, indicating that a string should follow, but there are no more operands.

System Action: The stage terminates with return code 156.

157E Null string found

Explanation: There are two consecutive delimiter characters.

System Action: The stage terminates with return code 157.

158E Modulo must be positive (it is *number*)

System Action: The stage terminates with return code 158.

159E Device *address* no longer exists

Explanation: Condition code 3 is received on an I/O operation to the device.

System Action: The stage terminates with return code 159.

161E 64K or more inbound data

Explanation: A 3270 generates 64K bytes or more of input data.

System Action: The stage terminates with return code 161.

User Response: If your terminal is a personal computer, the terminal simulator may have generated an incorrect inbound transmission.

162E Return code *number* from NUCEXT

Explanation: The return code shown is received when installing or retracting a nucleus extension.

System Action: Processing terminates with return code 162.

163E Missing keyword INPUT or OUTPUT

Explanation: SELECT and SEVER must have an operand.

System Action: Processing terminates with return code 163.

164E Direction "*word*" not valid or not supported

Explanation: A stage issues a pipeline command where the first operand is the word shown. This combination is not supported.

System Action: Processing terminates with return code 164.

165E Stream identifier *word* not valid

Explanation: A stage issues a pipeline command where *word* is expected to be a stream identifier. The combination shown is not supported.

System Action: Processing terminates with return code 165.

166E No real device attached for *device*

Explanation: The device driver requires a real device, but one is not attached.

System Action: The stage terminates with return code 166.

167E You cannot READ from the second reading station

Explanation: SELECT SECOND is in effect. The second reading station has no input stream associated and thus no record can be read.

System Action: The stage terminates with return code 167.

169E Stream identifier missing

Explanation: SELECT has no operands.

System Action: Processing terminates with return code 169.

170E Prefix or suffix type connector not allowed

Explanation: A pipeline specification that is issued with CALLPIPE contains an output connector at the beginning of a pipeline or an input connector at the end of a pipeline.

System Action: Processing terminates with return code 170.

User Response: Use the ADDPIPE pipeline command to process alternative input or redirect output.

172E Help not available for relative message number; issue PIPE HELP MENU for the Pipelines help menu

Explanation: The operand on PIPE HELP specifies a relative number for which no message is stored.

System Action: Processing terminates with return code 172.

173E No stage found to run

Explanation: CMS Pipelines has discovered an internal error. The pipeline is stalled, but error recovery finds no stage that is forced ready to run.

System Action: Processing terminates with return code 173.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in CMS Pipelines. Provide as documentation PIPDUMP LISTING created on the user's A disk.

174E Stream "identifier" is already defined

Explanation: The second component of the label refers to a stream that is already defined for the stage.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -174.

User Response: Choose another stream identifier for the label reference.

175E Language table not generated

Explanation: The language table describing message texts for multiple languages has not been generated in CMS Pipelines.

System Action: Processing terminates with return code 175.

176E Language "word" not found

Explanation: Messages for the requested language were not generated with CMS Pipelines.

System Action: Processing terminates with return code 176.

177I Spent number milliseconds in routine

Explanation: This message is issued when the message level includes the bit for 8K. A message is issued for each stage as it completes. Further messages are issued to list time spent in system services.

System Action: None.

178E Stream "identifier" is not found

Explanation: fanin is used with operands to designate a specific order of streams to be read, but the one shown cannot be selected. SELECT on spec requests a stream that is not defined.

System Action: The stage terminates with return code 178.

User Response: This error can be caused by a missing stage separator after fanin.

179E Character "char" is not an ASA carriage control character

Explanation: The file is not in the correct format.

System Action: The stage terminates with return code 179.

User Response: Check the input file.

180E Character X'hex' is not a machine carriage control character

Explanation: The file is not in the correct format.

System Action: The stage terminates with return code 180.

User Response: Check the input file.

181E PSW mask and key are X'hex', not X'FFE0' or X'03E0'

Explanation: With the bit for 2K on in the message level, the pipeline dispatcher finds that it is called from a program that is either disabled or executes in a key other than the one reserved for CMS user programs. The first two bytes of the PSW are substituted. They should be X'FFE0' in a 370-mode virtual machine, X'03E0' in an XA-mode virtual machine.

System Action: Control returns to the stage with return code 181. The function requested is not performed.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message 1 is issued to show the stage in error. Contact your systems support staff.

182W String "string" ignored in command

Explanation: An input operation is performed through the REXX interface. The pipeline command has more than two words.

System Action: Remaining words are ignored.

User Response: Ensure the pipeline command is issued correctly.

183E Output buffer overflow; number required

Explanation: While unpacking a file, a logical record is met that is longer than the maximum record length declared for the file in the first record.

System Action: The stage terminates with return code 183.

User Response: Check the input file. Use XEDIT to test if the input file is a proper packed file. Do not try to unpack with COPYFILE; it may cause a CMS ABEND.

184E Storage at address not released; R12 hex R14 hex

Explanation: A stage obtained storage. The area of storage was not released through the proper interface. The contents of general registers 12 and 14, at the time storage was allocated, are substituted in the message.

System Action: None.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message 1 is issued to show the stage in error. Contact your systems support staff.

System Programmer Response: Use NUCXMAP to locate the pipeline module. Determine if the module not releasing storage is part of *CMS Pipelines*.

185E Entry point name is not executable

Explanation: The entry point for the stage contains the operation code zero. Executing it would lead to a program check.

System Action: Message 1 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -185.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message 1 is issued to show the stage in error. Issue "pipe query" to display the level of *CMS Pipelines* that you are using; compare the response with the response in this book. Contact your systems support staff if this book applies to the level of *CMS Pipelines* that you are using and the program is a built-in one.

System Programmer Response: If *CMS Pipelines* is installed in a shared segment, ensure that sufficient space has been allocated.

186I PIPMOD MSGLEVEL number

Explanation: This is the response to "pipe query msglevel".

System Action: Return code 186 is reflected.

187E Keyword word must be LIFO or FIFO

Explanation: The operand to *stack* is not valid.

System Action: The stage terminates with return code 187.

189I Messages issued: list

Explanation: This message is issued or stacked when the message list is queried by PIPMOD QUERY MSGLIST. The list is 44 bytes long with four bytes for each message. Each message has a 3-three byte number and a 1-byte severity code. The entry for the last issued message is to the right. Zeros are provided in the leftmost entries when less than 11 messages have been issued since the pipeline module was initialised.

System Action: Processing continues. Return code 0 is reflected.

190E The character cannot begin a stage

Explanation: The first character of the definition of a stage is a special character.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -190.

191E Second character of connector not a period

Explanation: The first character is an asterisk, indicating a connector, but the second character is not a period.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -191.

192I ... Scan at position number; previous data "string"

Explanation: The number substituted is the number of characters from the beginning of the pipeline specification (including global options) to the current scan pointer. The last 20 characters before the scan pointer are substituted for *string*.

System Action: None.

User Response: The error is at or before the character indicated by the scan pointer.

193E Colon missing in connector

Explanation: The definition of a stage begins with an asterisk, but a blank character or a parenthesis is met before a colon.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -193.

194E Parenthesis not supported in connector

Explanation: A parenthesis is met in a connector.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -194.

195E Pipeline cannot contain only a connector

Explanation: A connector at the beginning of a pipeline ends the operand string, or it is followed by an end character.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -195.

196E Column ranges must be in ascending order and not overlapping

Explanation: *change* finds overlapping ranges or ranges that are not in ascending order from left to right.

System Action: The stage terminates with return code 196.

197E Range shorter than first string

Explanation: The “from” string is longer than a range.

System Action: The stage terminates with return code 197.

198E Count must be one when first string is null

Explanation: A null “from” string is present; the only acceptable count is one.

System Action: The stage terminates with return code 198.

200E Missing ending parenthesis in expression

Explanation: More left parentheses are met than can be paired with right parentheses in the expression.

System Action: The stage terminates with return code 200.

204E Too many ending parentheses in expression

Explanation: A right parenthesis is met for which there is no open left parenthesis.

System Action: The stage terminates with return code 204.

206E Expression missing

Explanation: An opening parenthesis is followed by a closing one or a comma; a comma is followed by a comma, or a comma is followed by a closing parenthesis.

System Action: The stage terminates with return code 206.

209E Segment length *number* not 2 or more

Explanation: The length byte in front of a segment is zero or one. This is not valid for data blocked in the netdata format.

System Action: The stage terminates with return code 209.

User Response: Check the input file. Ensure that the input stream is indeed in the netdata format and that records are padded to 80 bytes.

211E Second target missing

Explanation: A delimited string is found for the first target, but the second target is not present.

System Action: The stage terminates with return code 211.

212E Screen size *number* less than 1920 or greater than 16384

Explanation: The screen size (the product of the number of lines and columns) is less than the 1920 capacity of a model 2 screen, or larger than the 16384 positions addressable with 14-bit addressing.

System Action: The stage terminates with return code 212.

214E Mode *fm* is not accessed or not CMS format

Explanation: No CMS minidisk or directory is found with the mode letter shown. Either the mode is not accessed, the character is not a letter, or the disk is in OS format.

System Action: The stage terminates with return code 214.

215E File identifier "*file*" not complete or too long

Explanation: The identifier of a file to be looked up by *state* or *statew* is not two or three blank-delimited words.

System Action: The stage terminates with return code 215.

219E Input not in correct format (check word is "check word", not "word")

Explanation: A stage that expects a particular record from another stage did not read what it expected. For example, *outstore* finds an input record that does not describe a file in storage; or *tcpdata* reads a record that is not generated by *tcplisten*. "Input" should be taken to include records passed in a data space.

System Action: The stage terminates with return code 219.

User Response: Check the input file. Ensure that the correct stage is used to generate the file.

220E First record not a delimiter: "data"

Explanation: *maclib* finds that the first input record does not define the name of a member. The beginning of the first record is shown. Two double quotes with no substitution means that the first record is null. This is an error because the record belongs to no member.

System Action: The stage terminates with return code 220.

User Response: Check the input file. Ensure that the input stream has members delimited properly and that the operand, if used, is written correctly and in the correct case.

By default, members are delimited by "*COPY" starting in the first position of the record.

Note that the operand is not translated to upper case; write in upper case if the delimiters are upper case.

System Action: The stage terminates with return code 220.

221E Incorrect character "character" in expression

Explanation: The character shown is not valid in an expression.

System Action: The stage terminates with return code 221.

222E Secondary stream not defined

Explanation: Only the primary stream is defined. *update* requires two input and output streams. *lookup* requires at least two streams.

System Action: The stage terminates with return code 222.

User Response: *update* reads the master file from the primary input and writes the updated file to the primary output. Transactions (update control cards and replaced/inserted records) are read from the secondary input. The update log is written to the secondary output.

```
/* Sample update */
'pipe (end ?)',
  '< mstr fl|u:update|> m fl a',
  '?< upd fl|u:      |> u log a'
```

223E Sequence error in output file: previous to new

Explanation: A sequence error is introduced in the output file.

System Action: This message is written to the update log stream. Processing continues. Return code 8 is set unless other errors force a higher return code.

224E Premature end of primary input stream; sequence number number not found

Explanation: An update control record references the number shown, but it is not found before the input stream is exhausted.

System Action: This message is written to the update log stream. Processing continues. Return code 12 is set unless other errors force a higher return code.

225E Sequence number not found

Explanation: An update control record references the number shown, but it is not found. A line with a higher serial number is encountered.

System Action: This message is written to the update log stream. Processing continues. Return code 12 is set unless other errors force a higher return code.

226E Sequence field length length too long; 15 is maximum

Explanation: The length of the sequence field is larger than the maximum supported.

System Action: The stage terminates with return code 226.

227E Sequence field not present in record; number bytes read

Explanation: An input record is too short to contain the sequence field. All master input records are checked for this; detail records being inserted are checked if the control record indicates that the sequence field in the record is to be retained.

System Action: The stage terminates with return code 227.

User Response: Check the input file. Ensure that all records have a sequence field. Move the sequence field to the beginning of variable length records.

229E Sequence error in input stream from previous to new

Explanation: The input master file has a sequence error.

System Action: This message is written to the update log stream. Processing continues. Return code 8 is set unless other errors force a higher return code.

230E Unsupported format "type"

Explanation: The record format for the packed file is neither fixed nor variable.

System Action: The stage terminates with return code 230.

231E Null variable name

Explanation: The first two characters of an input record are the same.

System Action: The stage terminates with return code 231.

User Response: Check the input file. Ensure that a single delimiter is used to delimit the variable name from the data to load. The name must begin in the second column of the input record.

A blank or an asterisk (*) in column one indicates a comment line for which no variable is set.

232E Stem or variable name is too long; length is number bytes

Explanation: The variable name is too long. *var* supports at most 250 bytes for the variable name. *stem* supports at most 240 bytes in the name of the stem to allow for a 10-character sequence number.

System Action: The stage terminates with return code 232.

User Response: Choose a shorter name for the variable or stem.

233E No active EXECCOMM environment found

Explanation: A stage refers to the EXEC environment, but no such environment is found.

System Action: The stage terminates with return code 233.

User Response: Ensure that the pipeline is started from an EXEC when using filters referencing EXEC or REXX variables.

System Programmer Response: A NUCEXT for EXECCOMM received a nonzero return code.

234E Caller not REXX

Explanation: *rexxvars* is unable to obtain the interpreter private data. Most likely, an EXEC2 issued the PIPE command.

System Action: The stage terminates with return code 234.

User Response: Ensure that *rexxvars* is only called from REXX programs. Such programs begin with a REXX comment (*(* ... *)*).

235E Variable name is not valid: word

Explanation: The variable name is unacceptable to the EXECCOMM interface. The variable may be longer than 250 characters or it may contain a character that is not valid in a variable name.

System Action: The stage terminates with return code 235.

User Response: Ensure that the stem or variable name is spelt correctly. Do not put an ampersand (&) at the beginning of it. *varload* requires that the stem part of a variable name must be in upper case; a simple variable must be completely in upper case.

236E Too much data for variable name

Explanation: Too much data is to be set. The maximum length supported for EXEC2 variables is 255 bytes. This message is also issued when there is insufficient storage for EXECCOMM processing to complete.

System Action: The stage terminates with return code 236.

User Response: Use *chop* to truncate records if using EXEC2. With REXX, it is likely that you have run out of storage; it may help to increase virtual machine storage.

237E Error code X'hex' (return code number) from EXECCOMM

Explanation: *CMS Pipelines* is not prepared for the return code it receives from EXECCOMM.

System Action: Message 552 displays the EXECCOMM parameter list. The stage terminates with return code 237.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message 1 is issued to show the stage in error. Contact your systems support staff.

System Programmer Response: This is probably an error in *CMS Pipelines*. Consult the status codes defined in the member SHVBLOCK in DMSGPI MACLIB to see the meaning of the return code.

238E Record count "word" not zero or positive

Explanation: The contents of the variable that specifies the number of variables in a stemmed array cannot be converted to a number that is zero or positive. The name of this variable is formed by appending a zero (X'F0') to the operand to *stem*. The contents of the variable are substituted; results differ between REXX and EXEC2 when the variable is not "set" (when no assignment has been made to the variable). An unset EXEC2 variable is shown as null; REXX returns the name of an unset variable as its value.

System Action: The stage terminates with return code 238.

User Response: Remember to set the variable to an integer value before calling a subroutine pipeline using *stem* to read a stemmed array.

240E • 264E

Ensure that the variable set is the one referenced. Do not put an ampersand (&) at the beginning of a variable name when accessing variables in EXEC2 programs.

240E Function *name* not supported

Explanation: An expression has an identifier followed by a left parenthesis, indicating a function call, but the function requested does not exist.

System Action: The stage terminates with return code 240.

241E Record format or logical record length is not valid

Explanation: *members* is used with a file that is not fixed with 80 byte records; *qsam* is used for a file with a record format for which it is not designed.

System Action: The stage terminates with return code 241.

User Response: For *members*, correct the way the library is generated. For *qsam*, use a utility to change the record format to fixed or variable.

: 242E Too few arguments; *number* is minimum

: **Explanation:** Too few arguments were present for the function call.

: **System Action:** The stage terminates with return code 242.

: 243E Too many arguments; *number* is maximum

: **Explanation:** Too many arguments were present for the function call.

: **System Action:** The stage terminates with return code 243.

245W Operand *word* ignored

Explanation: You have specified options for *fullscr* that are incompatible amongst themselves or with the virtual machine architecture.

- Both operands CONSOLE and DIAG58 are specified.
- DIAG58 is specified in a virtual machine that is not in 370 mode.
- ASYNCHRONOUS and either or both of NOCLOSE or DIAG58 are specified.

. You have specified SENDTIME without specifying SENDDATE with *inmr123*.

System Action: Processing continues. The operand substituted is ignored.

250E Syntax error in expression

Explanation: A malformed expression is met. This includes adjacent operators, empty parentheses, and strings that are not separated by an operator.

System Action: The stage terminates with return code 250.

253E Data not a NETDATA control record

Explanation: A record is met that does not conform to the transmission data format. The beginning of the record is not X'E0' followed by 'INMR0' followed by a number 1 through 4, 6, or 7.

System Action: The stage terminates with return code 253.

User Response: Check the input file. Most likely, the input data is not from a file in the netdata format or data records have not been removed.

256I No pipeline specified on *pipe* command

Explanation: The PIPE command is issued without arguments.

System Action: The return code is 256.

User Response: Provide a pipeline specification with the PIPE command.

257E Subcommand environment *word* not found

Explanation: A device driver interfacing to a subcommand environment is unable to locate the requested environment.

System Action: The stage terminates with return code 257.

261E Unable to open *DDNAME*

Explanation: The third bit of DCBOFLGS stays zero.

System Action: The stage terminates with return code 261. A companion CMS message (DMSSOP036R) may have been issued.

User Response: The most likely cause is that no FILEDEF has been issued to define the data set. Ensure that DCB attributes are specified.

264E Too many streams

Explanation: Too many streams are defined for *merge*; a selection stage has more than two streams; a secondary stream is defined for a stage that does not use it.

System Action: The stage terminates with return code 264.

User Response: Cascade *merge* stages to merge the required number of streams. For other stages, this message usually indicates trouble with the multistream topology. For instance, this is a subroutine pipeline to select lines with A, B, or C:

```
'callpipe (end ? name ALLMSGs)',
'|*:',
'|a:locate string /A/',
'|f:faninany',
'|*:',
'?a:',
'|b:locate string /B/',
'|f:',
'?b:',
'| locate string /C/',
'|f:'
```

279E Tape identifier *word* not valid

Explanation: CMS gives return code 4 when the tape is read or written. This means that the device name is not valid.

System Action: The stage terminates with return code 279.

User Response: Valid operands are device addresses in the ranges 180 through 187 and 288 through 28F, and the operands TAP0 through TAPF.

280E Delimiter 16M or longer

Explanation: The argument string to *maclib* is longer than 16M.

System Action: The stage terminates with return code 280.

User Response: Choose a shorter token.

281W Mixed case command verb "*word*"

Explanation: *command* finds the first word to be different from its translation to upper case.

System Action: The tokenised parameter list is translated to upper case.

User Response: This message alerts you to the fact that results may be different on VM/System Product Releases 4 and 5. Write the command entirely in upper case to be sure the results are the same on the two releases of VM.

System Programmer Response: *command* inspects the operand defined by CD in SYSTEM KWDTABLE to see if it should translate the tokenised parameter list and issue this message. Use one of these values:

- 0 Issue no message and translate to upper case.
- 1 Issue message 281 and translate to upper case. This is the default.
- 2 Issue no message and leave the tokens in lower case.
- 3 Issue message 281 and leave the tokens in lower case.

282E Stage cannot be used with ADDPIPE

Explanation: One of the device drivers referring to REXX or EXEC variables is requested in a pipeline specification issued with ADDPIPE. Since the two programs would run in parallel, it is not possible to ensure that the EXEC COMM environment will remain for the duration of the new pipeline.

System Action: The stage terminates with return code 282.

User Response: Use CALLPIPE to load or store variables in a REXX filter.

283W Operand *word* ignored with *console*

Explanation: WAIT is specified for *fullscr*, and CONSOLE is specified or defaulted; WAIT is only valid with DIAG58.

System Action: Processing continues.

284E Field or string longer than 16M

Explanation: The first word in the argument to *maclib* is longer than 16M. Other stages may also require strings that are shorter than 16M.

It is more likely that there is an error in *CMS Pipelines*.

System Action: The stage terminates with return code 284.

287E Number *number* cannot be negative

Explanation: A negative number is specified for an operand to a stage that only supports zero or positive numbers.

System Action: The stage terminates with return code 287.

288I Posting ECB at *address*

Explanation: Information message issued when the dispatcher posts an ECB because the stage has no more streams connected. The option TRACE is active.

Note that posting an ECB is not traced in the normal course of events.

System Action: Processing continues.

289E Intervention required on *device*

Explanation: Intervention is required on the virtual device shown. On virtual unit record devices this may mean that system SPOOL space is full.

System Action: The stage terminates with return code 289. No message is issued to the system operator.

User Response: For a virtual unit record device, issue the CP command "ready" to make the device ready.

290E Tape address is write protected

Explanation: The tape does not have the write enable ring mounted. For an IBM 3480 or 3490 cartridge tape, the cartridge is protected against write.

System Action: The stage terminates with return code 290. No message is issued to the system operator.

User Response: Ensure the correct tape is mounted on the device. Ask the system tape operator to make the medium writable.

291E End of tape on device

Explanation: The tape has reached the end of the volume while writing a data record.

System Action: The stage terminates with return code 291.

User Response: Write a tape mark to the output tape. Mount another tape to continue. Use a control stage when writing more than one volume; the next record to write is available in the pipeline.

292E I/O error on address; CSW X'hex', CCW X'hex'

Explanation: An error occurred on the device.

System Action: The stage terminates with return code 292.

System Programmer Response: Note that this message is only issued for terminals that are locally attached; CP does not reflect the error on terminals connected through PVM, VM/VCNA, or VM/VTAM.

293I Sense data

Explanation: Sense data is available for the error reported with message 292. Sense bytes are normally shown in hexadecimal; a single sense byte having only one of the first six bits on is decoded:

CmdRej	Command reject. The virtual device does not support the operation. On a terminal, this may indicate that write structured field is not supported. A printer rejects punch command codes and <i>vice versa</i> .
IntvReqd	Intervention required. On a special GRAF, a dialled terminal has dropped. On unit record output, the CP command "notready" has been issued or the CP SPOOL system is full; issue the CP command "ready" to ready the device.
BusOutCk	Bus out check. The device detected a parity error on the data bus. This is an unlikely error condition, and it should not occur on a virtual unit record device.

EqpmtCk	Equipment check. An equipment malfunction is detected. This error is not likely to occur on a virtual SPOOL device.
DataCk	Data check. This error is not likely to occur on a virtual SPOOL device.
Overrun	The channel did not transmit data fast enough for the device. This error is not likely to occur.

System Action: None.

297E Return code number from diagnose X'A8'

Explanation: An error is reported on diagnose A8 to write unit record output.

System Action: The stage terminates with return code 297. Message 298s are written to list the contents of the parameter list to diagnose A8.

User Response: Inspect the sense data and determine the cause of error.

298I HCPSGIOP contents: hex

Explanation: The contents of the HCPSGIOP control block (used with diagnose A8) are listed in hexadecimal, 32 bytes at a time.

System Action: None.

300E Namelist does not end

Explanation: A left parenthesis is found opening a name list in a table definition, but no right parenthesis is found to close it.

System Action: The stage terminates with return code 300.

301E No position for last variable

Explanation: The argument string ends prematurely.

System Action: The stage terminates with return code 301.

302E Too many variable names specified (number); maximum is 254

Explanation: The ISPF maximum is exceeded.

System Action: The stage terminates with return code 302.

303E Return code number from function

Explanation: The return code shown is received when performing the ISPF or CMS function shown.

System Action: The stage terminates with return code 303.

304E ISPF is not active

Explanation: No ISPLINK subcommand environment is active, so it is not possible to access ISPF. Return code 1 is received when querying the subcommand environment ISPLINK.

System Action: The stage terminates with return code 304.

User Response: Ensure that ISPF is active.

System Programmer Response: Regenerate the pipeline module with the current ISPLINK TEXT if the ISPLINK interface has changed.

305E Table *word* is not open

Explanation: ISPF indicates with a return code 12 that the requested table is not open.

System Action: The stage terminates with return code 305.

User Response: Issue TBOPEN to open the table before running the pipeline referencing the table.

306E IUCV application *name* already active (HNDIUCV RC=4)

Explanation: The application name used by *CMS Pipelines* to establish IUCV connections is already known to CMS. This can happen when the PIPE command is entered recursively from a pipeline that uses *starmsg*.

User Response: Find out what is already connected to the system service.

System Action: The stage terminates with return code 306.

307E Unable to connect to *service*

Explanation: The path to the system service shown is severed rather than connected. For *MSG and *MSGALL this indicates that there is already a path connected to the service.

User Response: Find out what is already connected to the system service. This can be a *starmsg* stage in a pipeline that has invoked the PIPE command recursively, or it can be a different application, for instance full screen CMS.

System Action: Message 312 is issued if the user data field unless it contains all zero bits all one bits, or is blank. The path is severed. The stage terminates with return code 307.

308E CP system service *name* not valid

Explanation: Return code 1016 is received on a CMSIUCV macro. This indicates that the name of a CP system service is not valid.

System Action: The stage terminates with return code 308.

User Response: Check the first operand. *starmsg* is intended to connect to the services *MSG and *MSGALL.

309E This machine has too many IUCV connections

Explanation: Return code 1013 is received on CMSIUCV CONNECT.

System Action: The stage terminates with return code 309.

User Response: Contact your systems support staff.

310E Return code *number* from HNDIUCV

Explanation: CMS sets the code shown when a path to an IUCV service is declared.

System Action: The stage terminates with return code 310.

User Response: Return codes are listed in *z/VM CMS Macros and Functions Reference*, SC24-6262. Return code 4 indicates an attempt by two programs to access the same function.

311E Return code *number* from CMSIUCV *function*

Explanation: The return code shown was received from CMS when attempting to connect to a service.

System Action: The stage terminates with return code 311.

User Response: Return codes are listed in *z/VM CMS Macros and Functions Reference*, SC24-6262.

312I IPUSER: *hex*

Explanation: A path was severed. If the IPUSER field is neither blank nor zero, its contents are substituted. The substitution is a character string when the field consists entirely of printable characters; otherwise the field is displayed in hexadecimal.

313E IPRCODE *number* received on IUCV instruction

Explanation: The return code is not expected.

System Action: The stage terminates with return code 313.

314E Server *user ID* is not available

Explanation: Return code 1011 is received on CMSIUCV CONNECT. On CMS, the virtual machine is not logged on or has not enabled IUCV communications. On z/OS, no address space has connected to the VMCF subsystem for the name specified.

System Action: The stage terminates with return code 314.

315E Server has not declared a buffer

Explanation: Return code 1012 is received on CMSIUCV CONNECT.

System Action: The stage terminates with return code 315.

317E IUCV is not available to CMS

Explanation: The return code from HNDIUCV is 32.

System Action: The stage terminates with return code 317.

User Response: Contact your systems support staff to determine which product bypasses CMSIUCV.

318E Server machine has too many connections

Explanation: Return code 1014 is received on CMSIUCV CONNECT to the server machine.

System Action: The stage terminates with return code 318.

319E Not authorised to communicate with *service*

Explanation: Return code 1015 is received on CMSIUCV CONNECT. There is no IUCV statement in the directory authorising communication to the server.

System Action: The stage terminates with return code 319.

System Programmer Response: Use the IUCV ALLOWANY statement in the directory entry for the server virtual machine if anyone should be allowed to connect to it; use IUCV statements in the directory entries for individual users when you wish to authorise only some virtual machines to communicate with a server.

320E Unexpected IUCV interrupt with IPTYPE *type* on path *number*

Explanation: An IUCV interrupt is fielded where the type is not the expected one.

System Action: The stage terminates with return code 320.

324E CMSIUCV application not active in server

Explanation: The connection to the server was severed with user data all binary ones. For CMSIUCV this means that there is no active application by the name specified.

System Action: The stage terminates with return code 324.

333E System service *name* is in use

Explanation: The requested system service is already being used by a stage; it cannot be used by more than one stage at a time.

System Action: The stage terminates with return code 333.

User Response: Use *fanout* to create multiple copies of the output stream from *starmsg*.

334E FROM value not valid for file of size *number* records

Explanation: The FROM option on *diskslow* specifies a number that is larger than the number of records in the file when reading, or it specifies a number that is one larger than the number of records in the file when writing.

System Action: The stage terminates with return code 334.

335E Odd number of characters in hex data: *string*

Explanation: A prefix indicating a hexadecimal constant is found, but the remainder of the word does not contain an even number of characters.

System Action: The stage terminates with return code 335.

336E String length not divisible by 8: *string*

Explanation: A prefix indicating a binary constant is found, but the number of characters in the remainder of the word is not divisible by eight.

System Action: The stage terminates with return code 336.

337E Binary data missing after *prefix*

Explanation: A prefix indicating a binary constant is found, but there are no more characters in the argument string or the next character is blank.

System Action: The stage terminates with return code 337.

338E Not binary data: *string*

Explanation: A prefix indicating a binary constant is found, but the remainder of the word contains a character that is neither 0 nor 1.

System Action: The stage terminates with return code 338.

339E PIPSDEL return code *number*

Explanation: A return code is received on a conversion operation.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *CMS Pipelines*. Recreate the message with SET MSG ON to display the module that issues the message. Contact IBM for service.

340I IPARML: *message* (R0=*number*)

Explanation: The bits for 128 or 64 are on in the message level. The number is decoded when it represents a valid IUCV code.

341I .hex: *hex *char**

Explanation: Three lines are displayed for the IUCV parameter list and the ECB that are used for the request. Each line contains the hexadecimal storage address of the beginning of the data displayed. Up to 16 bytes are displayed in unpacked hexadecimal with character equivalents in EBCDIC.

342I Path *number* is connected to *service*

Explanation: The bit for 16 is on in the message level. A connection complete interrupt has been fielded.

343E IPAUDIT is not zero: *hex*

Explanation: The audit field in a message complete interrupt parameter is not all zero bits. If the audit field contains a single one bit for which there is a defined explanation, this is substituted in the message; otherwise the contents of the audit field are substituted. This indicates a programming error unless the audit field indicates that the message was rejected or the path was severed, in which case the cause is an action at the other end of the IUCV connection.

System Action: The stage terminates with return code 343.

User Response: Refer to “IUCV SEND” and “Message Complete External Interrupt” in *CP Programming Services*, SC24-6272.

344I IUCV External Interrupt *type*

Explanation: The bit for 32 is on in the message level. An external interrupt is being processed. The contents of the interrupt parameters are dumped from storage.

345E Originator *name* severed path *number*

Explanation: A connection pending interrupt was received from the virtual machine shown on the path shown. When accepting the connection, the CMSIUCV ACCEPT macro returns code 1020, indicating that the originator has severed the path in the meantime.

System Action: The stage terminates with return code 345.

346E No message found (id *number*)

Explanation: Condition code 2 is received on an IUCV instruction. This means that the message specified does not exist.

System Action: The stage terminates with return code 346.

347E Condition code 3 on IUCV instruction

Explanation: Condition code 3 is received on an IUCV instruction.

System Action: The stage terminates with return code 347.

348I UserData *data*

Explanation: An IUCV service severed the path unexpectedly. TCP/IP will indicate the reason in the user data field. The user data field is neither all zeros nor all ones. If it is printable the contents are shown as sixteen characters, otherwise the contents are shown as thirty-two hexadecimal characters.

350E Primary key longer than secondary

Explanation: The primary key is longer than the secondary key.

System Action: The stage terminates with return code 350.

352E Input record is *number* bytes; it should be *number*

Explanation: The input record does not have the length required for the function.

System Action: The stage terminates with return code 352.

User Response: *fntfst* is intended to process the output from *state* NOFORMAT.

354E Return code *number* from SQL, detected in module *module*

Explanation: A negative return code is received from SQL.

System Action: The stage terminates with the return code shown. Messages 355, 356, and 369 are issued to describe the error further.

User Response: Try the command “pipe help sqlcode” to see if it is possible to obtain the information about the return code from SQL. Refer to the section on SQLCODES in *SQL/Data System Messages and Codes for IBM VM Systems*, SH09-8079 if online help fails.

355I ... RDS: *number* DBSS: *number*; *number* rows done; *string*

Explanation: This message is issued after message 354 to display additional information from the SQL communications area.

System Action: None.

System Programmer Response: The numbers are obtained from the SQL communications area. *string* shows the flags; blanks have been changed to minus to maintain alignment.

356I ... Message parameter *string*

Explanation: The SQL communication area has a parameter string with one or more items in it; each is listed in a separate message.

System Action: None.

357E SQL RC -934: Unable to find module *module*; run SQLINIT

Explanation: SQL is unable to initialise.

System Action: The stage terminates with return code -934.

User Response: The most likely cause is that the SQL interface modules are not generated on your A disk. Issue "Filedef * clear" followed by "sqlinit db(sqldba)" to create the modules SQL uses to find the resource manager. Specify the name of the database in the SQLINIT command. Be sure to access the minidisk that contains the SQL parameters (normally SQLDBA 195). Contact your systems support staff if there is no SQLINIT EXEC available to you or if there is no ARIRVSTC TEXT available to you.

358E SQL RC -805: Access module *name* not found; refer to help for SQL to generate access module

Explanation: The access module is not generated.

System Action: The stage terminates with return code -805.

User Response: Contact your systems support staff.

System Programmer Response: An access module must be generated before *CMS Pipelines* can access SQL. The recommended approach (which is the way *CMS Pipelines* is shipped) is to generate the access module as 5785RAC.PIPSQI and then grant the use of that to everyone ("grant run on 5785rac.pipsqi to public").

Use PGMOWNER to specify a program owner for a particular invocation of *sql*.

SQL/Data System Application Programming for IBM VM Systems, SH09-8086 describes how to use SQLPREP to generate an access module.

359E SQL object already exists

Explanation: SQLCODE -601 is received, indicating that the object you tried to create is already known to SQL.

System Action: The stage terminates with return code -601.

360E Table *table* does not exist

Explanation: SQLCODE -204 is received, indicating that SQL is not able to find the table in its catalogues.

System Action: The stage terminates with return code -204.

361I ... SQL processing: *string*

Explanation: An error occurred in a SQL statement. The statement is shown.

System Action: None.

362E DESCRIBE followed by "*word*"; must be SELECT

Explanation: You must provide the select statement you wish described.

System Action: The stage terminates with return code 362.

User Response: Use the operand SELECT to designate the beginning of the query.

363E SQL RC -205: Column *name* not found in *creator.table*

Explanation: SQL indicates that a column is not present in a table.

System Action: The stage terminates with return code -205

364E Unable to obtain help from SQL (return code *number*)

Explanation: A nonzero return code is obtained when reading the index to the SQL return code information in SQLDBA.SYSTEXT1..

System Action: The stage terminates with return code 364.

User Response: The error reported is likely to be -934 or -806, which indicate that you have not identified the SQL virtual machine or that the access module for *CMS Pipelines* has not been generated. Additional messages are likely to be issued; refer to help for the message issued.

365E SQL has no information about *topic*

Explanation: *help* is processing a help request for the SQL topic shown. The tables are successfully selected, but the query result is null. This means that there is no information available about the topic.

System Action: The stage terminates with return code 365.

User Response: Ensure that the correct return code is put in the query. Use "pipe help sqlcode" to display help for the last SQLCODE received by *sql*.

366E Too few input streams

Explanation: EXECUTE is used to perform SQL commands. The primary input stream. has more *sql* INSERT statements without values() than there are additional input streams defined.

System Action: The stage terminates with return code 366.

User Response: Provide the input for the first INSERT on the secondary input stream. The primary input stream is read for additional statements; it is not available for data.

367E Use SQL CONNECT TO to identify the subsystem
(Reason *hex*)

Explanation: *sql* receives return code 12 from DSNALI. The most likely reasons are that the database is not the default DSN or that you are not authorised to use the plan PIPSQI with the resource you are connected to.

System Action: The stage terminates with return code 367.

User Response: Use *sql* CONNECT TO to specify the subsystem name that you wish to connect to. This specification remains in effect until the end of the PIPE command.

368E 10 SQL stages already active

Explanation: There are already 10 *sql* stages active.

System Action: The stage terminates with return code 368.

User Response: Try to change the pipeline topology to make some *sql* stages complete before starting others.

369I ... SQL statement prepared: *string*

Explanation: An error is reported by SQL while it is processing a dynamically prepared statement. The statement is substituted.

370E Cursor has been closed

Explanation: SQL code -504 is received while a cursor is used to read a line of a query or insert a line. The most likely cause is that another *sql* stage has committed the unit of work or rolled it back.

System Action: The stage terminates with return code -504.

User Response: Ensure that all concurrent *sql* stages specify NOCOMMIT. Use a buffer stage to separate a query from the stage processing the result.

Use a subroutine pipeline to ensure that a query is processed correctly before the result is processed further; direct the result to a stemmed array where it can be referenced by a second pipeline after the return code for the first one is tested and found OK.

371E ARIRVSTC TEXT is not available; run SQLINIT

Explanation: The object module that contains the SQL bootstrap code is not linked into the pipeline module, nor is it accessible as a file.

System Action: The stage terminates with return code 371.

User Response: Issue "Filedef * clear" followed by "sqlinit db(sqldba)" to create the modules SQL uses to find the resource manager. Specify the name of the database in the SQLINIT command. Be sure to access the minidisk that contains the SQL parameters (normally SQLDBA 195). Contact your systems support staff if there is no SQLINIT EXEC available to you or if there is no ARIRVSTC TEXT available to you.

373E No SQL stub module or DB2 not present in system

Explanation: On CMS, the entry point ARIRVSTC is not resolved. On z/OS, the module DSNALI could not be loaded. This indicates that DB2 is not installed in the system.

System Action: The stage terminates with return code 373.

User Response: Contact your systems support staff.

System Programmer Response: Make sure the pipeline module is linked with the SQL interface module if you do not wish the interface code to be loaded dynamically.

```
pipgmod arirvstc.text
rename nxpipe module a pipeline = =
nucxdrop pipmod
```

374E DB2 connection using plan *word* already active

Explanation: The option PLAN is specified, but a different plan is already in use.

System Action: The stage terminates with return code 374.

375E DB2 already connected to subsystem *word*

Explanation: The option SSID is specified, but a different subsystem is already in use.

System Action: The stage terminates with return code 375.

376E Return code *number* reason *hex* from call to DSNALI

Explanation: The return code (register 15) and reason code (register 0) substituted are received in response to a call to CAF OPEN.

System Action: The stage terminates with return code 376.

377E Subsystem *word* is not defined

Explanation: DSNALI returns reason code X'00F30006', which means that the subsystem identification is not valid (or more likely not defined).

User Response: Contact your database administrator to determine the subsystem id to specify or contact your systems support staff to generate the correct default in *TSO Pipelines*.

System Programmer Response: The system keyword QZ defines the default subsystem identifier. This is DSN by default.

System Action: The stage terminates with return code 377.

378E Plan *word* is not authorised

Explanation: DSNALI returns reason code X'00F30034', which means that the user is not authorised for the plan name substituted.

System Action: The stage terminates with return code 378.

379E Subsystem name is not up

Explanation: DSNALI returns reason codes X'00F30002', X'00F30011', or X'00F30012'. These indicate that the DB2 subsystem is not up.

System Action: The stage terminates with return code 379.

380E Left parenthesis missing

Explanation: A left parenthesis is expected for a list of items, but one is not found.

System Action: The stage terminates with return code 380.

381E Right parenthesis missing

Explanation: A left parenthesis for a list of items has been met, but no right parenthesis is found.

System Action: The PIPE command or stage terminates with return code 381.

382E Nothing specified within parentheses

Explanation: An opening parenthesis is found with only blank characters before the closing parenthesis.

System Action: The stage terminates with return code 382.

391E Unsupported conversion type

Explanation: The type shown is syntactically correct to request a conversion of a field, but the conversion is not available. An example of such conversion is B2F.

System Action: The stage terminates with return code 391.

User Response: Use two *spec* stages to perform the conversion via an intermediary format; for instance, character.

392E Conversion error in routine 2: type, record 3: number (reason code 1: reason); data: "4: string"

Explanation: The string shown has a value that is not valid for the conversion requested.

System Action: The stage terminates with return code 392.

User Response: Check the input file.

The naming conventions for the conversion routines are adopted from REXX. The formats of the input and output types are defined by the characters surrounding the number '2':

- C A character string with the internal representation of the data type. In C2D, for instance, the input character string should be four characters corresponding to a fullword integer in two's complement notation.
- X A zoned hexadecimal string containing the digits 0 through 9 and the letters a through f (in upper case or lower case, or a mixture). There must be an even

number of hexadecimal digits in the string; blanks are only allowed at byte boundaries.

- D A zoned decimal integer made up from the digits 0 through 9, possibly with a leading sign. Leading and trailing blanks are allowed, as are blanks between the sign and the number.
- B A string of zeros and ones. The length must be a multiple of eight.
- F A floating point number. Examples of floating point numbers are -5, .03, 2.7e-76.
- V A variable length character field that has a halfword (two bytes) length prefix.
- P A zoned decimal number made up from the digits 0 through 9 with an optional leading sign and an optional decimal point. Leading and trailing blanks are allowed, as are blanks between the sign and the number.
- I A date. If the field contains six characters, it is taken to be year, month, and date (two digits each). When the field is eight characters or longer, it consists of two characters century followed by six characters date followed by an optional timestamp which can contain up to six digits. The timestamp contains three two-digit fields for hours, minutes, and seconds.

The reason code describes what went wrong:

- 4 Missing character in number or exponent.
- 8 A character in number or exponent is not valid.
- 12 Exponent overflow or underflow.
- 16 Incorrect character in integer or number too large for fullword representation.
- 20 The input field for C2D is longer than 4 bytes and the sign is not propagated.
- 24 The first or last character of a hexadecimal field is blank.
- 28 Odd number of characters in a hexadecimal field.
- 32 Incorrect character in a hexadecimal field.
- 36 The number of characters in a bit field is not divisible by 8.
- 40 Character in a bit field is neither 0 nor 1.
- 44 Floating point number is shorter than 2 bytes or longer than 8 bytes.
- 48 A field to be converted to varying character is 64K or longer; the length cannot be expressed as a halfword integer.
- 52 The length of a varying character field is longer than the input field available.
- 56 A number consists of blanks or a sign. That is, it contains no digits.
- 60 Incorrect character (not a decimal digit) in number to be packed.
- 64 A packed decimal field contains a sign that is not valid (it is a digit).
- 68 A packed decimal field contains a digit that is not valid (it is a sign).
- 72 A packed decimal field is null.

- 76 A Julian date is shorter than six characters, it has an odd number of characters, or it is longer than fourteen characters.
- 80 A Julian date that is eight characters or longer begins with two digits that are less than 19.
- 84 Incorrect digit in Julian date (not decimal).
- 88 Month or day is zero or too large.
- 92 A field to be converted to Julian is shorter than three bytes or longer than seven bytes.
- 96 A field to be converted to Julian does not contain X'F' in the rightmost four bits of the third or fourth byte.
- 100 A field to be converted to Julian contains a character that is not valid (it is not not decimal).
- 104 A field to be converted to Julian contains a value that is not valid (century field over X'80', which means beyond year 9999; day larger than 365/366; hours, minutes, seconds out of range).
- 108 Hours are larger than 23; minutes or seconds are larger than 59.

393E Output field too short to contain field length

Explanation: V2C conversion is requested with an explicit output field length. The length is less than 3, which means that no characters can be loaded in the field.

System Action: The stage terminates with return code 393.

400E Delay word is not acceptable

Explanation: The first word of an input record is not three (or fewer) decimal numbers separated by colons.

System Action: The stage terminates with return code 400.

401E Input record too short (number bytes)

Explanation: For *asmcont*, an input record after a statement indicating continuation is shorter than 16 bytes, which means that there is no continuation text. For *join* KEYLENGTH, the input record was shorter than the specified key length.

For *lookup* SETCOUNT, the input master record is shorter than 10 bytes, which means that it does not contain a full count field. Likewise, for *lookup* INCREMENT, the input detail record is shorter than 10 bytes and thus cannot contain the increment field. For *fbawrite*, the record is shorter than 24 bytes.

System Action: The stage terminates with return code 401.

User Response: Check the input file. If you are using *asmcont*, ensure that the input file is indeed an Assembler file.

402I Calling Syntax Exit

Explanation: Pipeline dispatcher trace is active. The stage is defined with a syntax exit which is called.

System Action: None.

405E Minimal C program tries to extend DSA

Explanation: A program using a minimal C runtime has run out of stack space.

System Action: The program terminates. The stage returns with code 405.

User Response: Use the C systems programmer environment for the program. This makes it look like any other Assembler program.

406E Unsupported language code number for entry point

Explanation: The eleventh byte in the entry point table entry for the stage has an unsupported code.

System Action: The stage terminates with return code 406.

User Response: Write no more than three words before the first asterisk on a line in an entry point table unless you wish to select a high level language interface.

407E PLISTART or CEESTART is not present

Explanation: A PL/I or C program is requested in the entry point table, but no such program is linked into the module.

System Action: The stage terminates with return code 407.

User Response: Inspect the fourth word in the entry point table used to resolve the program to see if the specified high level language is indeed the intended one.

409E Assert failure code at address

Explanation: CMS Pipelines has discovered an internal error. A program check operation exception is forced to indicate a condition which should not occur.

System Action: Message 411 is issued if the information is available. CMS ABEND processing continues.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message 1 is issued to show the stage in error. Make a note of the code and the following message. Contact your systems support staff.

System Programmer Response: Investigate whether corrective service is available.

410E ABEND code at address; PSW hex

Explanation: A CMS ABEND has occurred in the main pipeline module. The ABEND code indicates the type of failure. The immediate CMS command HX causes ABEND 222.

The PSW at time of ABEND (ABNPSW) is substituted. The contents of storage locations 140-143 (Program Interruption Identification) are displayed after the PSW when bit 12 of the PSW is one. This field is meaningful only if a program check caused the ABEND.

System Action: Message 411 is issued if the information is available. CMS ABEND processing continues.

411I • 505E

User Response: Ensure the message level is odd (it is unless you have changed it) and that message 1 is issued to show the stage in error. Make a note of the information for your systems support staff. Contact your systems support staff.

System Programmer Response: Investigate whether corrective service is available.

411I ... In procedure; offset *offset* in module

Explanation: Informational message issued at ABEND when it can be determined in which module the failure occurred.

System Action: Message 412 is issued four times to display the contents of the general registers at the time of failure.

User Response: Note the information and provide it along with information in messages 409 and 410.

412I ... GPR*n*: *hex*

Explanation: The contents of the general registers at the time of failure are displayed.

User Response: Note the information and provide it along with information in messages 409 through 411.

413I ... Store *hex*: *hex*

Explanation: The contents of storage at the point of failure are substituted.

420E Return code *number* reading or writing block *number* on disk *mode*

Explanation: The return code shown is received when reading a block from the disk.

System Action: The stage terminates with return code 420.

User Response: Ensure that the block is within the disk extents. Ensure that the disk is formatted correctly. Ensure that the block number(s) are in decimal.

421E File mode *string* more than one character

Explanation: A word in the argument string to *adtfst* is longer than one character.

System Action: The stage terminates with return code 421.

User Response: Write each mode letter as a blank-delimited word when more than one mode letter is processed.

498E Output descriptor *name* is not valid

Explanation: Reason code X'035C8002' was received when dynamically allocating a SYSOUT data set. The output descriptor contains a character that is not valid.

System Action: The stage terminates with return code 498.

499E Output descriptor *name* is not defined

Explanation: Reason code X'04CC8002' was received when dynamically allocating a SYSOUT data set.

System Action: The stage terminates with return code 499.

500E Data set *DSNAME* is partitioned

Explanation: The requested data set is partitioned but no second operand is provided to indicate a specific member.

System Action: The stage terminates with return code 500.

User Response: Select a specific member when allocating the data set.

501E No data set is allocated for *DDNAME*

Explanation: There is no data set allocated for the data definition name shown. The return code 4 is received on the RDJFCB macro.

System Action: The stage terminates with return code 501.

502E Member *name* already selected by allocation

Explanation: A second operand is found to indicate that a member of a partitioned data set is to be read or written, but the specific member name substituted is specified in the allocation of the data set.

System Action: The stage terminates with return code 502.

User Response: Allocate the complete partitioned data set when referring to members.

503E Return code *number* obtaining data set control block

Explanation: The return code from OBTAIN is greater than 8. Return code 12 indicates an error reading the volume table of contents. Return code 16 indicates a programming error in *CMS Pipelines*.

System Action: The stage terminates with return code 503.

504E Data set *DSNAME* does not exist

Explanation: Return code 4 or 8 is received when trying to locate the data set with OBTAIN, which indicates that the volume is not mounted or that the data set does not exist.

System Action: The stage terminates with return code 504.

505E Data set *DSNAME* is not partitioned

Explanation: A member is requested and the data set control block does not indicate partitioned organisation.

System Action: The stage terminates with return code 505.

506E DDNAME *name* is permanently concatenated

Explanation: *qsam* does not support permanent concatenations.

System Action: The stage terminates with return code 506.

User Response: Use > to specify the particular data set into which the member should be stored.

507E Member *name* not found

Explanation: FIND or BLDL gives return code 4, indicating that the requested member is not in the data set.

System Action: The stage terminates with return code 507.

508E Output descriptor too long: *word*

Explanation: The word is longer than 26 characters. This is the limit for an output descriptor.

System Action: The stage terminates with return code 508.

509E Unacceptable spool file identifier *SFID*

Explanation: For *reader*, the number shown is negative; for *xab*, the number is negative or larger than 64K.

System Action: The stage terminates with return code 509.

510E Spool ID *SFID* not found or incompatible with reader

Explanation: For *reader*, the file cannot be ordered; CP returns condition code 2 on diagnose 14 subcode X'C', which means that the file does not exist, is in hold, does not have the class that the reader is spooled for, or is open on another reader.

For *xab*, CP gives return code 44, indicating that the SPOOL file does not exist.

System Action: The stage terminates with return code 510.

511E Spool file identifier *SFID* rejected by CP

Explanation: CP gives return code 44 on the diagnose instruction to manipulate the external attribute buffer.

System Action: The stage terminates with return code 511.

512E Virtual device *device* not a spooled printer

Explanation: The virtual device type class is not unit record output.

System Action: The stage terminates with return code 512.

513E Return code *number* reading or writing XAB (parameters *hex*)

Explanation: CP gives the return code shown on the diagnose instruction used to perform the function you have requested. The two fullwords of RY and RY+1 are displayed.

System Action: The stage terminates with return code 513.

User Response: Refer to the error description of diagnose codes B4 and B8. Diagnose B4 is used to set or read the external attribute buffer of a virtual printer; diagnose B8 is used with a SPOOL file.

514E Record length *number* is over the maximum 32767

Explanation: The first input record is longer than the maximum allowed.

System Action: The stage terminates with return code 514.

User Response: Check the input file.

515E Not a decimal range: *word*

Explanation: A decimal number or range is expected but the word shown is found.

System Action: The stage terminates with return code 515.

516E Not a record number or a range of record numbers: *word*

Explanation: Though an acceptable range of decimal numbers, the word shown cannot represent a range of records. The beginning of the range is zero or less, or the end of the range is less than the beginning.

System Action: The stage terminates with return code 516.

517E Record *number* not present in file

Explanation: The record requested is not in the file.

System Action: The stage terminates with return code 517.

518E Record *number* truncated

Explanation: The record requested has been replaced or added since *diskrandom* obtained information about the file from CMS.

System Action: The stage terminates with return code 518.

530E Destructive overlap

Explanation: A record is moved or appended to a buffer. Condition code 3 is set, indicating destructive overlap. This can be caused by indiscriminate use of *storage*.

System Action: The stage terminates with return code 530.

531E Word must be 8 characters; it is *number*

Explanation: An operand to *optcdj* is present, but not eight characters.

System Action: The stage terminates with return code 531.

532E Storage key *hex* not acceptable

Explanation: The third operand to *storage* or the input to *adrspc* CREATE does not designate an acceptable storage key. For *storage*, storage key zero is not accepted and the rightmost four bits of the key must be zero. In all cases, the three leftmost bytes must be zero.

System Action: The stage terminates with return code 532.

User Response: Specify E0 for the user storage key; specify F0 for the nucleus key.

533E Storage at *address* is protected

Explanation: *storage* gets a program check code 4 (protection) while loading data into the storage area.

System Action: The stage terminates with return code 533.

534E Storage at *address* is not addressable

Explanation: *storage* gets program check code 5 (addressing).

System Action: The stage terminates with return code 534.

535E Program check *code*

Explanation: *storage* gets a program check that is neither protection nor addressing.

System Action: The stage terminates with return code 535.

536E Buffer header destroyed: *hex*

Explanation: *CMS Pipelines* has discovered an internal error. The pointer to the next available byte is below the base address of the buffer.

System Action: The stage terminates with return code 536.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message 1 is issued to show the stage in error. Contact your systems support staff.

System Programmer Response: This is likely to be an error in *CMS Pipelines*.

537I Commit level *number*

Explanation: Pipeline dispatcher trace is active. The stage commits to the level shown.

System Action: None.

538I Query state of *side stream stream*

Explanation: Pipeline dispatcher trace is active. The stage requests the status of the stream on the side shown.

System Action: None.

539E Do not connect unused *side stream stream*

Explanation: A stream is connected that the stage does not use. This is often a symptom of an incorrect placement of a label reference.

A selection stage (for instance, *find*) detects that the secondary input stream is connected. *collate* detects that the tertiary input stream is connected. *fanin*, *faninany*, *merge*, and *overlay* detect a connected output stream other than the primary one. *fanout* detects a connected input stream other than the primary one. *lookup* detects that input stream 4 is connected.

System Action: The stage terminates with return code 539.

User Response: Ensure that the reference to the label that specifies the secondary output stream for a selection stage is after an end character.

540E Command is longer than 132 (*number characters*)

Explanation: A command to *vmc* is longer than the maximum of 132 characters.

System Action: The stage terminates with return code 540.

541E VMCF is in use by another stage

Explanation: *vmc* cannot use VMCF because another stage is using it.

System Action: The stage terminates with return code 541.

542E Unable to communicate with *user ID*

Explanation: CP sets return code 5 when *vmc* tries to communicate with the server.

System Action: The stage terminates with return code 542.

543E Return code *number* from VMCF: *string*

Explanation: CP sets the return code shown on a VMCF request.

System Action: Message 544 is issued to display the parameter list. The stage terminates with return code 543.

544I VMCPARMS: *hex*

Explanation: The parameter list is displayed for the request that CP rejects.

System Action: None.

545E VMCF message rejected by user *user ID*

Explanation: The VMCF message is rejected by the user ID shown.

System Action: The stage terminates with return code 545.

546E Input record length *number* is too short; 11 is minimum

Explanation: *diskupdate* reads a record that is too short to have a record number prefix (columns 1 through 10) and one byte of data to write to the file.

System Action: The stage terminates with return code 546.

547E Record number *number* is beyond end-of-file

Explanation: *diskupdate* receives return code 7 writing the record. This means that the file has variable record format and that the record number is larger than one plus the number of records in the file.

System Action: The stage terminates with return code 547.

User Response: Specify fixed format to create sparse files.

548I SEVER function requested for *side*

Explanation: Pipeline dispatcher trace is active. The stage severs the connection on the side shown.

System Action: None.

549E Return code *number*, reason code *number*, R0 hex from IRXINIT

Explanation: The return code and reason code shown are received when trying to find the environment for the REXX program that issued a pipeline specification with Address link or Address attach. The reason code is valid only when the return code is 20.

User Response: Refer to the IRXINIT return and reason codes in *TSO Extensions Version 2 REXX Reference*, SC28-1883.

System Action: The stage terminates with return code 549.

System Programmer Response: Reason code 24 means that the environment table has too few entries for the number of concurrent REXX programs that the user wishes to run. Refer to *TSO Extensions Version 2 REXX Reference*, SC28-1883.

550E Unable to access variables

Explanation: The TSO service routine gives return code 40, indicating that there is no active CLIST environment.

System Action: The stage terminates with return code 550.

552I SHVBLOCK: *hex*

Explanation: The EXECOMM parameter list is displayed on two lines.

553E Return code *number* calling IRXSUBCM function

Explanation: The return code shown is received.

System Action: The stage terminates with return code 553.

554E Stream identifier *string* must not be numeric

Explanation: A stream identifier in a label declaration or label reference is numeric.

System Action: Message 192 is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -554.

User Response: Begin a stream identifier with a letter. Stream numbers are assigned when labels are referenced. You cannot specify that a particular stream is to have a particular number; use a stream identifier to refer to a stream without knowing its number.

555I Issue PIPE AHELP PIPE or PIPE AHELP MENU

Explanation: "pipe ?" was issued. This message tells you where to go for help.

System Action: Return code 0 is set.

556E Asterisk cannot end output column range

Explanation: An output column range ends with an asterisk.

System Action: Return code 556 is set.

User Response: Write a single column to put a field at a particular position, extending as far as required. Use a range to put the field into a particular range of columns, padding or truncating as necessary. Use the operand NEXT instead of a range or column number to abut the field to the contents of the output buffer; this is equivalent to the concatenate operator (||) in REXX. Use the operand NEXTWORD instead of a range or column number to append a blank and the field to the output record built so far. (The blank is suppressed if the record is empty.)

557E Not authorised to obtain CP load map

Explanation: A program check is reflected on the diagnose 38 that is issued to read the CP symbol table.

System Action: The stage terminates with return code 557.

User Response: Ensure that the virtual machine has command privileges to issue diagnose 38. By default, privilege class C or E is required; your installation may have changed the privilege classes in an override file.

558E No symbol table available

Explanation: Condition code 1 is set on the diagnose 38, indicating that there is no CP symbol table available.

System Action: The stage terminates with return code 558.

559E Paging error reading symbol table

Explanation: Condition code 3 is set on the diagnose 38, indicating that CP is unable to read the symbol table.

System Action: The stage terminates with return code 559.

560I CMS Pipelines, 5741-A07 level *hex*

Explanation: The fullword version identifier is substituted. The first digit is the version number, currently 1. The second digit is the release number, currently 1. The third and fourth digit is the modification level, currently 12. The last four digits are the serial number within the modification level.

System Action: Return code 0 is set.

561E File *file* is no longer in storage

Explanation: The file shown was found to be in storage (return code 0 from EXECSTAT) when the pipeline specification was scanned. By the time the file is to be read, EXECSTAT no longer gives return code zero.

System Action: The stage terminates with return code 561.

User Response: Investigate whether a REXX program, which has started at commit level -1, has dropped the file from storage.

562E Alternate exec processor *name*; return code *number*

Explanation: The REXX compiler runtime library is not available if the return code shown is -3.

System Action: The stage terminates with return code 562.

563W ANYOF assumed in front of *string*

Explanation: A delimited string that contains more than one character is specified without a keyword to specify how to interpret it. It is most likely that you wish this interpreted as a string rather than as an enumerated list of characters. This message is suppressed if the delimited string contains one character; the question is clearly moot.

User Response: Use the keyword ANYOF to specify a delimited string of characters enumerating characters that match a single character position in the input record. Use STRING to specify that the target is a string of characters that must occur in the sequence shown to match.

564W Range(s) should be before keyword; put more than one in parentheses

Explanation: A range is specified after the keyword. The order should be reversed.

565W Stage is obsolete; use *name* instead

Explanation: A stage is used that will be retracted.

User Response: Use the name for the stage that will continue to be available.

Old	New
<i>cpasis</i>	<i>cp</i>
<i>countlms</i>	<i>count</i> LINES
<i>xtract</i>	<i>members</i> . Add TXLIB * between the file name and the list of members.

566W Use secondary output instead of stack

Explanation: *count* specifies an option to put the result on the program stack.

User Response: Connect the secondary output stream and process the result without using the program stack. This does not disturb the contents of the stack and does not expose you to problems with multiple *count* stages.

568I PL/I: *message*

Explanation: The PL/I runtime environment has called the message server to issue the message substituted.

569E Path to *service* severed (path *number*)

Explanation: The path to the system service shown was unexpectedly severed. A connection complete interrupt was received previously for the path.

System Action: The path is severed. The stage terminates with return code 569.

570E Unexpected IUCV interrupt with IPTYPE *type* on path *number*

Explanation: An IUCV interrupt is fielded while the stage is waiting for a connection complete or path severed interrupt. This represents a CP/CMS IUCV protocol error.

System Action: The path is severed. The stage terminates with return code 570.

571E Virtual device *device number* is in use by another stage

Explanation: Two stages try to operate the same device concurrently.

System Action: The stage terminates with return code 571.

572E Unable to load file (EXECLOAD return code number)

Explanation: In the syntax exit for a REXX filter, CMS indicates that the program is disk resident. When the time comes to run the program, EXECLOAD fails with the return code shown.

System Action: The stage terminates with return code 572.

573E Last text unit or GDF order not complete

Explanation: The length field of a text unit specifies a count that is larger than the number of bytes remaining in the input record. A GDF order specifies more data than remain in the record.

System Action: The stage terminates with return code 573.

574E Address is odd

Explanation: The entry address specified with *runat* is odd.

System Action: The entry point is not resolved.

575E Block padded with hex; it should be X'00'

Explanation: A GDF structured field ends in the penultimate position of an input block. The last byte of the block should be zero, but it contains the data shown.

System Action: The stage terminates with return code 575.

576E Input record is number bytes; disk block size is number bytes

Explanation: The length of an input record is not zero or 10 plus the disk block size.

System Action: The stage terminates with return code 576.

577E Return code number from STIMERM

Explanation: A nonzero return code is received on the macro to set a timer interval on z/OS. This message is most likely to be the result of a programming error in *CMS Pipelines*.

System Action: The stage terminates with return code 577.

User Response: Refer to the return codes for the STIMERM macro instructions in *MVS/ESA Application Development Reference: Services for Assembler Language Programs*, GC28-1642.

579E Return code number from DYNALOC; reason hex

Explanation: The error number shown is returned from dynamic allocation. The error number and reason code are substituted.

System Action: The stage terminates with return code 579.

User Response: Refer to the section "SVC 99 Return Codes" in *MVS/ESA Application Development Guide: Authorized Assembler Language Programs*, GC28-1645.

580I DDNAME allocated: word

Explanation: Display the DDNAME allocated to the data set. This message is issued only if the bit for 1024 is turned on in the message level.

581E >> cannot append to a member

Explanation: >> does not support a member name.

System Action: The stage terminates with return code 581.

582E Incorrect DSNAME "string"

Explanation: The data set name is not well formed (null or missing ending quote) or SVC 99 gives error code X'035C' for the data set name.

System Action: The stage terminates with return code 582.

583E Incorrect member name "string"

Explanation: The member name is not well formed (null or no opening parenthesis) or SVC 99 gives error code X'035C' for the member name.

System Action: The stage terminates with return code 583.

584I Enter PIPESTOP, PIPESTALL, or immediate pipeline command

Explanation: You have entered the *TSO Pipelines* attention exit for the first time. If you enter the command PIPESTOP or enter a null line and hit attention again, all stages that wait on an ECB will be signalled to terminate. This is likely to bring the pipeline to a halt.

System Action: None.

585I ECBs posted: number; hit attention again to stall the pipeline

Explanation: You have entered the *TSO Pipelines* attention exit for the second time. If you hit attention again, the pipeline will be stalled. This will terminate the pipeline unless a stage is in a loop.

System Action: All stages waiting on an external event (waiting on an ECB) are signalled to terminate.

586I Hit attention again to terminate waiting stages

Explanation: You have entered the *TSO Pipelines* attention exit for the first time. If you hit attention again, all stages that wait on an ECB will be signalled to terminate. This is likely to bring the pipeline to a halt.

System Action: None.

587E Immediate command *name* is not active

Explanation: An immediate command was entered, but no *immcmd* stage is active for this command.

System Action: The command is ignored. A subsequent attention will cause *TSO Pipelines* to terminate those stages that wait on an external event.

590E User data length is over 62 or odd (it is *number*)

Explanation: Explicit user data to STOW with a member in a partitioned data set is either too long or it contains an odd number of characters.

System Action: The stage terminates with return code 590.

591E Return code *number* reason code *hex* from BLDL

Explanation: The return code shown was received when searching for a member in the directory of a partitioned data set. The contents of register 0 (the reason code) are substituted in hexadecimal.

System Action: The stage terminates with return code 591.

592E Conflicting allocation for data set *DSNAME*

Explanation: Dynamic allocation sets return code 02100002, which indicates that the data set is already allocated with a disposition that conflicts with the one requested. < and *pdsdirect* allocate DISP=SHR; > allocates DISP=OLD; >> allocates DISP=MOD.

System Action: The stage terminates with return code 592.

593E Shared data set *DSNAME* cannot be allocated exclusive

Explanation: Dynamic allocation sets return code 020C0000, which indicates that a request for exclusive allocation of a shared data set was rejected.

User Response: Ensure you wish to modify the data set. Use the SHR operand to indicate that a shared allocation should be used.

System Action: The stage terminates with return code 593.

594E Return code *number* reason code *hex* from STOW

Explanation: The return code shown was received when adding a member to the partitioned data set. The contents of register 0 (the reason code) are substituted in hexadecimal.

System Action: The stage terminates with return code 594.

595E Member name is not allowed for this function

Explanation: The program does not support a member name.

System Action: The stage terminates with return code 595.

596E Data set name too long: *name*

Explanation: The data set name plus the prefix (if active) is longer than forty-four characters.

System Action: The stage terminates with return code 596.

597E Member name or generation too long in *DSNAME* *name*

Explanation: The argument contains a left parenthesis, indicating that a generation number or a member is present. There are more than eight characters to the end of the argument.

System Action: The stage terminates with return code 597.

598E Null member name or generation in *DSNAME* *name*

Explanation: The argument contains a left parenthesis, indicating that a generation number or a member is present, but no further characters are present.

System Action: The stage terminates with return code 598.

599E Null *DSNAME* *name*

Explanation: The argument consists of a single quote or two quotes, or the first character is a right parenthesis for the beginning of a member. This is not a valid *DSNAME*.

System Action: The stage terminates with return code 599.

600E Return code *number* from TGET

Explanation: The return code shown is received when reading the terminal.

System Action: The stage terminates with return code 600.

601E Return code *number* from STFSMODE

Explanation: Full screen mode is not set.

System Action: The stage terminates with return code 601.

602E Unsupported data set organisation *hex*

Explanation: The data set organisation is neither physical sequential nor partitioned. The DSORG field is substituted.

System Action: The stage terminates with return code 602.

603E Unable to read directory for member *name*

Explanation: FIND gives a return code that is neither zero nor four.

System Action: The stage terminates with return code 603.

604E Null DDNAME

Explanation: The argument begins with the keyword DDNAME=, but there are no further characters or the next character is a left parenthesis to indicate a member.

System Action: The stage terminates with return code 604.

605E DDNAME longer than 8 characters: *word*

Explanation: The argument begins with the keyword DDNAME=; it is followed by a word that is more than eight characters.

System Action: The stage terminates with return code 605.

606E Null member name in DDNAME *name*

Explanation: The argument contains a left parenthesis, indicating that a member is present, but no further characters are present.

System Action: The stage terminates with return code 606.

607E Member name too long in DDNAME *name*

Explanation: The argument contains a left parenthesis, indicating that a member is present. There are more than eight characters to the end of the argument.

System Action: The stage terminates with return code 607.

608E Incorrectly specified DSNAME *word*

Explanation: A generation data group number in parentheses is followed by a character that is not a left parenthesis.

System Action: The stage terminates with return code 608.

609E ABEND code reason code *number*

Explanation: The DCB ABEND exit is driven for the abnormal termination condition substituted.

System Action: The ABEND condition is reset. The stage terminates with return code 609.

611E Cannot set CONSOLE exit

Explanation: *fullscr* ASYNCHRONOUS cannot set the console exit routine because the path turned out to be opened already.

User Response: Do not specify a path; let *fullscr* assign one.

System Action: The stage terminates with return code 611.

612I Parmlist: *hex*

Explanation: The contents of the EXECCOMM parameter list are substituted.

613E Pipeline specification is not issued with CALLPIPE

Explanation: The PRODUCER is requested, but the stage is not in a pipeline specification that has been issued with CALLPIPE. Thus, the integrity of the requested variable pool cannot be ensured.

System Action: The stage terminates with return code 613.

614E Caller's current input stream is not connected

Explanation: The PRODUCER is requested and the stage is in a pipeline specification that has been issued with CALLPIPE, but the caller's currently selected input stream is not connected. Thus, there is no producer stage and hence no variable pool to select.

System Action: The stage terminates with return code 614.

615E Caller's producer is not connected to caller

Explanation: The PRODUCER is requested, the stage is a pipeline specification that has been issued with CALLPIPE, and the caller's currently selected input stream is connected, but the output stream from the stage has been reconnected, or the stage has selected another output stream. Thus, input records do not correlate with the variable pool requested.

System Action: The stage terminates with return code 615.

616E Caller's producer is not blocked waiting for output

Explanation: The PRODUCER is requested, the stage is in a pipeline specification that has been issued with CALLPIPE, the caller's currently selected input stream is connected, and the output stream from the stage is connected to the caller, but the stage is not waiting for an output operation to complete. Thus, the integrity of the variable pool cannot be ensured.

System Action: The stage terminates with return code 616.

617E File does not have fixed format records; do not specify keyword

Explanation: The keyword BLOCKED is specified for a file that has variable length records. CMS does not support blocked read of such a file.

System Action: The stage terminates with return code 617.

620W Unsupported code page number

Explanation: A code page number is requested that *xlate* does not support.

System Action: The code page number is ignored.

621W Impossible target string

Explanation: The target string is longer than the column range in which to look for the string; no input record can ever be matched.

System Action: None.

622E Mask and string are not the same length

Explanation: The two delimited strings specified for MASK are not the same length.

System Action: The stage terminates with return code 622.

623E Unrecognised relational operator word

Explanation: A relational operator is expected, but not found. The valid operators are: ==, !=, <<, <=>, >>, >>=, EQ, NE, LT, LE, GT, and GE.

System Action: The stage terminates with return code 623.

624E Premature end of expression

Explanation: An operator or left parenthesis is met at the end of the expression. The expression is not complete.

System Action: The stage terminates with return code 624.

625E Target expression missing

Explanation: A keyword (for instance TO) is met, indicating that a target should follow, but there are no more arguments.

System Action: The stage terminates with return code 625.

626E Target data missing for keyword

Explanation: A keyword (for instance RECORD) is met, indicating the type of target to match, but there are no more arguments.

System Action: The stage terminates with return code 626.

627E Null program read from stream

Explanation: The program list contains no lines.

System Action: The stage terminates with return code 627.

635E Option word conflicts with option word

Explanation: Two incompatible options are specified.

System Action: The stage terminates with return code 635.

636E Error in encoded pipeline specification; reason code number

Explanation: Pass 1 of the scanner found a syntax error in a pipeline specification. This is an error in *CMS Pipelines*.

System Action: The stage terminates with return code 636.

User Response: Contact your systems support staff.

System Programmer Response: Report which built-in program issues the message, its argument string, and the reason code. The reason codes are:

- 8 The level of the encoded pipeline block is higher than supported by the version of *CMS Pipelines* that is being used.
- 4 A bit is on other than the ones for ADDPIPE and CALLPIPE.
- 1 Null pipeline. A pipeline begin item is after another pipeline begin item.
- 2 Stage after end connector. A connector that is not at the beginning of the pipeline has been processed. It is followed by a stage item rather than a pipeline begin item.
- 3 Label after ending connector. A connector that is not at the beginning of the pipeline has been processed. It is followed by a label reference item rather than a pipeline begin item.
- 4 Blank label reference. The label field of a label reference item has a leading blank.
- 5 More than one ending connector. A connector that is not at the beginning of the pipeline has been processed. It is followed by another connector item rather than a pipeline begin item.
- 6 Bad input parameter list. Unrecognised item code. Most likely the item list is not terminated properly.
- 7 Incomplete pipeline. The last pipeline has no stages, no label reference, and at most one connector.
- 8 The specification does not begin with a pipeline begin item.

637E Return code number on IDENTIFY for entry point

Explanation: z/OS sets the return code shown when *TSO Pipelines* attempts to identify the entry point shown.

User Response: Ensure that the PIPE command is called correctly; the module must be invoked or loaded in a way that allows the entry point to be identified.

System Programmer Response: Investigate whether the entry point name is already in LINKPACK or JOBPACK.

System Action: The stage terminates with return code 637.

638I SVC 99 parameter list hex

Explanation: Message level 1024 is on or dynamic allocation indicates an error in the parameter list. The parameter list is displayed.

639E Scaling allowed with packed data only

Explanation: A conversion routine is requested and a left parenthesis follows immediately. This is valid only when converting to or from packed decimal.

System Action: The stage terminates with return code 639.

640I Text unit type data

Explanation: The six bytes type/count/length field is substituted followed by the contents of the first data field. If the data are entirely printable, they are shown as characters; otherwise they are shown in hexadecimal.

641I Last connected output stream severed by its consumer

Explanation: Tracing is active for the stage. All output streams are now severed. The last output stream was severed by its consumer, rather than by the stage.

642E ZONE already specified

Explanation: The keyword ZONE is specified with *zone* or it is specified twice with *casei*.

System Action: The stage terminates with return code 642.

643E HLASM not found in storage

Explanation: The High Level Assembler module (HLASM) was loaded into storage, but the PROGMAP command did not provide information about the module.

System Action: The stage terminates with return code 643.

644E Timestamp word not valid; reason code number

Explanation: An ISO timestamp is not valid. The input record must contain the year (four digits) followed by five fields containing month, day, hour (24 hour clock), minute, and second (two digits each). It may be followed by one to six decimal digits representing a fraction of a second.

The reason code shows which test has failed:

- 4 The input record is shorter than 14 characters or longer than 20 characters after stripping leading and trailing blanks.
- 8 Year is not a number or the number is less than 1900.
- 12 Month is not a number, it is not positive, or it is greater than 12.
- ! 16 Day is not a number, it is not positive, or it is greater than the number of days in the month specified.
- ! 20 Hour is not a number, it is negative, or it is greater than 23.
- 24 Minute is not a number, it is negative, or it is greater than 59.
- 28 Second is not a number, it is negative, or it is greater than 59.
- 32 Fraction is not a number, it is negative, or it is greater than 999999.

System Action: The stage terminates with return code 644.

650E CP system service word is in use by another program

Explanation: CP severs a connection request to the system service substituted. This indicates that the service is already connected by some program that does not run under control of *CMS Pipelines*.

System Action: The stage terminates with return code 650.

651E DCSS word is not loaded

Explanation: An attempt was made to connect to *MONITOR using the segment name substituted. The monitor severed the connection with error code X'18', indicating that the segment was not available.

System Action: The stage terminates with return code 651.

652E DCSS name word does not match the DCSS name already established

Explanation: An attempt was made to connect to *MONITOR using the segment name substituted. The monitor severed the connection with error code X'28', indicating that some other virtual machine is already connected to the monitor using a different segment name.

User Response: Issue the CP command "monitor query" to determine the name of the segment currently in use.

System Action: The stage terminates with return code 652.

653E Monitor is currently running in shared mode; exclusive request rejected

Explanation: An attempt was made to connect to *MONITOR for exclusive use of the monitor segment. The monitor severed the connection with error code X'34', indicating that some other virtual machine is already connected to the monitor in shared mode.

System Action: The stage terminates with return code 653.

654E Monitor is currently running in exclusive mode; shared request rejected

Explanation: An attempt was made to connect to *MONITOR for shared use of the monitor segment. The monitor severed the connection with error code X'38', indicating that some other virtual machine is already connected to the monitor in exclusive mode.

System Action: The stage terminates with return code 654.

655E Not a named saved segment: word

Explanation: An attempt was made to connect to *MONITOR using the segment name substituted. The monitor severed the connection with error code X'3C', indicating that the substituted word is not the name of a discontinuous shared segment; it could, for instance, be a named saved system.

System Action: The stage terminates with return code 655.

656E Connection to word severed with code word

Explanation: A connection request to a system service was rejected.

User Response: Refer to the documentation for the system service shown.

System Action: The stage terminates with return code 656.

657E Limit of connections to word is reached

Explanation: A connection request to a system service was rejected with return code X'0C', indicating that the maximum number of connections supported for this service has already been reached.

System Action: The stage terminates with return code 657.

658E Too many concurrent STIMERM requests

Explanation: Return code X'1C' is received on a timer request. This indicates that 16 requests are already pending for the task. The other timer requests can be issued by *delay* stages or by host commands run through, for example, *command*.

System Action: The stage terminates with return code 658.

659E Return code number from LINEWRT macro

Explanation: The return code shown was received on a LINEWRT macro. Return code 104 means that there was insufficient storage to complete the request. Return code 24 means that the parameter list built by *CMS Pipelines* is rejected by CMS.

User Response: For return code 24, contact your systems support staff to report the problem.

System Action: The stage terminates with return code 659.

660E Unsupported code page number

Explanation: A FROM or TO was met, but the following word does not represent a supported code page number.

System Action: The stage terminates with return code 660.

661E Please ask nicely

Explanation: The *dmsabend* built-in program did not find the appropriate argument string for it to cause an ABEND.

User Response: Do not try to force ABENDS in *CMS Pipelines* unless you have been instructed to do so by IBM.

System Action: The stage terminates with return code 661.

662E Environment already specified (keyword is met)

Explanation: A number or one of the keywords MAIN or PRODUCER has already been specified to designate the environment to use. The keyword that is substituted is met later in the operand list.

System Action: The stage terminates with return code 662.

663E Unable to generate delimiter for variable name

Explanation: The name of the variable and the characters declared as beginning a comment (these characters are not eligible to be delimiter characters) contain between them all 256 possible values for a byte. Thus it is impossible to generate a delimiter character to be used to delimit the name of the variable.

User Response: Specify a shorter comment string.

System Action: The stage terminates with return code 663.

664E Keyword is not supported when stage is first: word

Explanation: The program is used as a first stage of a pipeline. The operand is valid only in a stage that is not first in a pipeline.

System Action: The stage terminates with return code 664.

665E Exponent is not valid: *word*

Explanation: A numeric constant is being scanned. The letter "E" is met. Either there is no number after the letter or the value of the exponent overflows a 32-bit integer.

System Action: The stage terminates with return code 665.

666E Syntax error in expression; reason code *number*

Explanation: The expression is not syntactically correct. The number describes the error:

- Internal error (negative length remains to be scanned).
- 0 Unexpected character at the beginning of an expression or after (.
- 1 A digit is expected for the number of a counter, but something else was found.
- 2 A counter was scanned; it was not followed by an operator or a).
- 3 An identifier or an expression has been scanned; it was not followed by an operator or a). Note that assignment operators cannot be immediately to the right of identifiers or expressions.
- 4 ! not followed by =.
- 5 Assignment attempted to something that is not a counter.
- 6 A vertical bar is not followed by another one to make up the logical OR operator. Be sure to use four vertical bars if they are also stage separators This self-escapes them down to two bars that are seen by *specs*.
- 7 An ampersand is not followed by another one to make up the logical AND operator.
- 100 Unpaired colon (:).
- 101 Two consecutive question marks (?). Use parentheses to group a conditional expression between the ? and the : of a containing one.

System Action: The stage terminates with return code 666.

667E Arithmetic overflow

Explanation: The result of evaluating an expression or an intermediary result is beyond the range that can be represented.

System Action: The stage terminates with return code 667.

668E Divisor is zero

Explanation: Division by zero is attempted.

System Action: The stage terminates with return code 668.

670E Picture longer than 255 characters: *picture*

Explanation: The word following PICTURE contains more than 255 characters.

System Action: The stage terminates with return code 670.

671E Unacceptable character *character in picture picture*

Explanation: The character is not one of the valid characters.

System Action: The stage terminates with return code 671.

672E Unacceptable picture *picture; word is incorrect (reason code number)*

Explanation: An incorrect sequence of picture characters is found.

User Response: Compare the contents of the word substituted with the contents of the picture string to see where in the string the error was detected.

System Programmer Response: The reason code should be reported when calling IBM for service. It reflects the internal state of the finite state machine that is used to decode the picture; the encoding is unspecified; it might change as a result of corrective service or new function being added.

System Action: The stage terminates with return code 672.

673E Picture has more than one V: *picture*

Explanation: Only one V character is allowed in a picture.

System Action: The stage terminates with return code 673.

674E Unacceptable drifting sign in picture *picture*

Explanation: A drifting sign character is not the same as the original sign character.

System Action: The stage terminates with return code 674.

675E Unacceptable zero suppress/protect in picture *picture*

Explanation: A zero suppress or currency protect character is not the same as the previous one.

System Action: The stage terminates with return code 675.

676E No digits selected in picture *picture*

Explanation: A leading sign is found in a picture, but no digits are selected.

System Action: The stage terminates with return code 676.

677E No exponent digits in picture *picture*

Explanation: The letter E is met, but no digit select characters follow.

System Action: The stage terminates with return code 677.

678E More than fifteen exponent digits in picture *picture*

Explanation: The letter E is met followed by more than fifteen digit selectors. The exponent can contain at most ten digits.

System Action: The stage terminates with return code 678.

679E Exponent too large: *number*

Explanation: The exponent has more significant digits than the picture allows. The exponent is substituted.

System Action: The stage terminates with return code 679.

680E Record length is zero

Explanation: The first byte of a logical record contains binary zeros. This is not valid, because the minimum record length is one (a record that contains a byte count of one and no data).

System Action: The stage terminates with return code 680.

681E Input record length (number) is over the maximum allowed (*number*)

Explanation: An input record is longer than the maximum allowed.

System Action: The stage terminates with return code 681.

User Response: Check the input file.

682I TXTunit list *hex*

Explanation: Message level 1024 is on or dynamic allocation indicates an error in the parameter list. The list of pointers to text units is displayed.

683I STAX return code *number*

Explanation: A nonzero return code is received on a STAX macro. The attention exit is not established.

System Programmer Response: Note the conditions under which this message is issued and report the problem to IBM if *TSO Pipelines* is being used in a supported environment.

684E Unsupported system variable *word*

Explanation: *sysvar* receives a syntax error when it tries to obtain the variable.

System Action: The stage terminates with return code 684.

685E OpenExtensions is not available (reason code number)

Explanation: *CMS Pipelines* is unable to call a service routine to access an OpenExtensions file. The reason code indicates which particular test failed:

- 1 The CVT field CSRTABLE does not contain a pointer. It contains -1. (Offset 544, decimal).

- 2 The resource slot in the OpenExtensions callable services function table points to a dummy entry.
- 3 The OpenMVS slot in the CSR table does not contain a pointer. It contains zero. (Offset 24, decimal).
- 4 The table of entry points to OpenExtensions callable services is too short to contain the function requested. That is, the operating system does not support the function requested.
- 5 The table of entry points to OpenExtensions callable services contains a dummy entry for the function requested. That is, the operating system does not support the function requested or the function is not available in this particular configuration.
- 10 The CMS is a release where the simulated CVT is too short to contain the pointer to the CSRTABLE.
- 4 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed. It is either zero or negative.
- 5 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed. It does not point within the virtual machine.
- 6 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed, or the CVT has been corrupted. The byte at offset X'74' has the bit for X'40' zero. This bit indicates that the system is CMS.
- 7 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed, or the CVT has been corrupted. The byte at offset X'74' has one or more of the bits for X'3F' nonzero. These bits must be zero on CMS.

User Response: Contact your systems support staff if the reason code indicates that the CVT pointer is destroyed. These reason codes are issued on CMS only.

System Programmer Response: Investigate if corrective service is available. In particular, ensure that the fix for APAR VM61261 is applied.

System Action: The stage terminates with return code 685.

686E OpenExtensions return code number reason code
hex function: word

Explanation: A call to the OpenExtensions failed.

The return code is shown as a decoded errno if the value is one of those recognised by *CMS Pipelines*; otherwise the return code is shown as a decimal number.

The reason code is shown in hexadecimal. Only the last four digits are significant.

The function is shown as the name of the equivalent POSIX function.

- ! **User Response:** For CMS, refer to *OpenExtensions Callable Service Reference, SC24-6203*. For z/OS, refer to *OS/390 Messages and Codes* for the error codes. In earlier releases, the codes were listed in the appendices of SC23-3020, *UNIX System Service Programming: Assembler Callable Services Reference*. Appendix A describes the

return codes; Appendix B describes the reason codes; and Appendix C describes the offsets.

System Action: The stage terminates with return code 686.

687E Relational operator expected; found *word*

Explanation: A relational operator is expected, but the word is not a supported one.

User Response: Note that the operators are the “strict” operators:

```

== Equal.
!= Not equal
<< Less than.
<<= Less than or equal.
>> Greater than.
>>= Greater than or equal.

```

For example, a single equal sign is not a supported relational operator.

688I CSW *hex*; last CCW *hex*; some data *hex*

Explanation: The channel status word is displayed.

689E Workstation file is missing: *word*

Explanation: Error code 110 is received when the file is opened by the server program.

System Action: The stage terminates with return code 689.

690E Logical drive was not found: *word*

Explanation: Error code 15 is received when the file is opened by the server program.

System Action: The stage terminates with return code 690.

691E Directory is missing: *word*

Explanation: Error code 3 is received when the file is opened by the server program.

System Action: The stage terminates with return code 691.

692E No diskette in drive: *word*

Explanation: Error code 21 is received when the file is opened by the server program.

System Action: The stage terminates with return code 692.

693I Packages sent: *number*; packages received: *number*

Explanation: This message is issued when *os2file* discovers an error after at least one package has been exchanged with the workstation server program.

694E Pipeline is not called from a driving program

Explanation: *fitting* is invoked in a pipeline set that has not been initialised for fittings. Thus, the stages have nothing with which to interface.

System Action: The stage terminates with return code 694.

695E Fitting already defined: "*name*"

Explanation: *fitting* is issued in a pipeline set that already has a fitting of that name defined.

System Action: The stage terminates with return code 695.

699E Return code *number* from function (file: *word*)

Explanation: An error was returned from the communication device. This could be a result of a programming error in *CMS Pipelines* or in the device driver that processes the request on the work station.

System Action: The stage terminates with return code 699.

700E File descriptor *number* is not open (reason code *hex*)

Explanation: Return code 113 (EBADF) is received on a request to read or write a file. The reason code further describes the error condition.

System Action: The stage terminates with return code 700.

701E File or directory does not exist (path "*string*" reason code *hex*)

Explanation: Return code 129 (ENOENT) is received on a request to open a file. The reason code further describes the error condition.

System Action: The stage terminates with return code 701.

702I ... Parameter: *hex*

Explanation: The bit for message level 1024 is on and an error was reported for a call to OpenExtensions. The first eight bytes of each parameter are shown.

703I Opening "*hex*"

Explanation: The bit for message level 1024 is on. A file in the hierarchical file system is being opened.

704E A component of path is not a directory (path "*string*", reason code *hex*)

Explanation: Return code 135 (ENOTDIR) is received on a request to open a file. The reason code further describes the error condition.

System Action: The stage terminates with return code 704.

705E Last character is a slash (path "string")

Explanation: Return code 129 (ENOENT) and reason code X'0109' (JREndingSlash0Creat) is received on a request to open a file.

System Action: The stage terminates with return code 705.

706E File system is quiescing (path "string")

Explanation: Return code 129 (ENOENT) and reason code X'018F' (JRQuiescing) is received on a request to open a file.

System Action: The stage terminates with return code 706.

707E Component in path name is too long: path "string"

Explanation: Return code 126 (ENAMETOOLONG) with reason code X'003E' (JRCompNameTooLong) is received on a request to open a file. A component (file name) is longer than NAME_MAX (255).

System Action: The stage terminates with return code 707.

708E Path name is too long: path "string"

Explanation: Return code 126 (ENAMETOOLONG) with reason code X'0039' (JRPathTooLong) is received on a request to open a file. A path name is longer than PATH_MAX (1023). This can be a result of the substitution of symbolic links.

System Action: The stage terminates with return code 708.

709E Unsupported file type number (path "string")

Explanation: The file is neither a regular file nor a FIFO. The actual file type is substituted:

- 1 Directory.
- 5 Symbolic link.
- 6 Block special file.

System Action: The stage terminates with return code 709.

710E Unsupported file type number (file descriptor "number")

Explanation: The file is neither a regular file nor a FIFO. The actual file type is substituted:

- 1 Directory.
- 5 Symbolic link.
- 6 Block special file.

System Action: The stage terminates with return code 710.

711E Function not supported: word

Explanation: The first word of an input record to *hfsxecute* or *hfsquery* is not a supported one.

System Action: The stage terminates with return code 711.

712E Path name is missing from the input record

Explanation: An input record to *hfsxecute* contains only one word.

System Action: The stage terminates with return code 712.

713E Mode is not valid: word

Explanation: The second word of an input record to *hfsxecute* does not contain a valid mode specification. The mode contains one to four octal digits.

System Action: The stage terminates with return code 713.

714E Unacceptable interval word

Explanation: The word is not proper for seconds with an optional fraction of microseconds. There may be at most six digits after the period; no component of the number may be negative; and the number must be positive.

System Action: The stage terminates with return code 714.

715E Not octal: word

Explanation: A leading zero is found, but one of the digits is not in the range from zero to seven. The second character is not "x", which would indicate a hexadecimal value.

System Action: The stage terminates with return code 715.

716E Not a dotted decimal network address: word

Explanation: A word that begins with a digit is scanned for a network address, but the word does not conform to the dotted decimal notation defined for `inet_addr()`. A component could be too large; or there could be more than three dots.

System Action: The stage terminates with return code 716.

717I Ignoring IUCV interrupt for message number; waiting for number (interrupt on path number; sent on number)

Explanation: One of the device drivers for TCP/IP received an IUCV interrupt for a message that it does not have outstanding with TCP/IP.

System Action: The interrupt is ignored.

System Programmer Response: If the two path numbers are different, the interrupt is being processed by the wrong stage. Investigate whether corrective service is available.

718I Returning to application

Explanation: A copipe is returning to the application program.

719I Resuming pipeline

Explanation: The application program has resumed the copipe with a request.

720I Terminating pipeline

Explanation: The application program has resumed the copipe without a request parameter list.

721I RPL hex

Explanation: The Request Parameter List is displayed.

722I Resolved fitting identifier

Explanation: A Request Parameter List is paired with an active *fitting* stage.

723I Fitting identifier not resolved

Explanation: A Request Parameter List was not paired with an active *fitting* stage. No current stage is active for the fitting.

724I Posting fitting identifier with hex

Explanation: A Request Parameter List is paired with an active *fitting* stage and it has work to do.

725I Returning to the pipeline dispatcher

Explanation: All Request Parameter Lists have been processed and the *fitting* stages posted to wake up.

726I No RPLs changed state

Explanation: All Request Parameter Lists have been processed, but none changed state. Thus, the application has cheated.

System Action: The status code is set accordingly.

727I string

Explanation: A tracing message.

728I number description

Explanation: Statistics are requested. The contents of a counter are displayed.

729I Letting dispatcher wait

Explanation: The application has indicated that it does not wish to regain control until a particular *fitting* stage has produced or consumed a record.

730E No data sets found matching *DSNAME*

Explanation: The return code from LOCATE was eight. This indicates that no matching entries were found.

User Response: The leading qualification should not end with a period.

System Action: The stage terminates with return code 730.

731E Return code *number* from SVC 26

Explanation: The return code from LOCATE was neither zero nor eight.

System Action: The stage terminates with return code 731.

732E Return code *number* from DMSCSL

Explanation: The callable services interface returned a return code that was not expected.

System Action: The stage terminates with return code 732.

733E Return code *number* reason code *number* from routine

Explanation: The callable services interface returned a nonzero return code. The return code, reason code, and routine name are substituted.

System Action: The stage terminates with return code 733.

734E CSL Routine *name* is not loaded

Explanation: Return code -7 was received from Callable Services.

User Response: Contact your system support staff to investigate whether the callable services have been set up correctly for your virtual machine.

System Programmer Response: It is an error in *CMS Pipelines* if this message is issued on releases prior to VM/ESA Version 1 Release 2.0.

System Action: The stage terminates with return code 734.

735E Callable Services are not available

Explanation: Return code -12 was received from Callable Services.

User Response: Contact your system support staff to investigate whether the callable services have been set up correctly for your virtual machine.

System Programmer Response: It is an error in *CMS Pipelines* if this message is issued on releases prior to VM/ESA Version 1 Release 2.0.

System Action: The stage terminates with return code 735.

736E Too few parameters in call to *name* (number found)

Explanation: Return code -11 was received from Callable Services.

User Response: Contact your system support staff.

System Programmer Response: This is a programming error in *CMS Pipelines*. Investigate whether corrective service is available.

System Action: The stage terminates with return code 736.

737E Too many parameters in call to *name* (number found)

Explanation: Return code -10 was received from Callable Services.

User Response: Contact your system support staff.

System Programmer Response: This is a programming error in *CMS Pipelines*. Investigate whether corrective service is available.

System Action: The stage terminates with return code 737.

738E Router did not resolve entry point

Explanation: The routine to resolve the entry point returned a value of zero.

User Response: Contact your system support staff.

System Programmer Response: This is a programming error in *CMS Pipelines* or an error in generating PIPELINE MODULE (DMPIPE MODULE on VM/ESA). Investigate whether corrective service is available.

System Action: The stage terminates with return code 738.

740E File "*words*" does not exist or you are not authorised for it

Explanation: Reason code 90220 was received from Callable Service DMSEXIST.

System Action: The stage terminates with return code 740.

741E Record format "*character*" is not supported

Explanation: The record format is neither F nor V. A blank indicates OS-format file; a hyphen indicates that the file is migrated.

System Action: The stage terminates with return code 741.

742E Incorrect file "*file*" (reason code number)

Explanation: The file name and file type are both an asterisk; or reason codes 90420, 90430, 90445, 90450, or 90455 were returned by DMSEXIST. A component of the file identification is longer than eight characters, contains an asterisk, or a percent sign. Reason code zero is set when the file name is an asterisk and the file type is an asterisk. For information about nonzero reason codes, refer to DMSEXIST in *CMS Application Development Reference*, SC24-5451.

System Action: The stage terminates with return code 742.

743I File "*file*"

Explanation: Open failed for a file. The file name parameter is shown.

744I Open flags *words*

Explanation: Open failed for a file. The open flags parameter is shown.

745E Existing record length is not *number*

Explanation: Reason code 90121 is received when overwriting a record of a variable record format file.

System Action: The stage terminates with return code 745.

746E File *file* is open with incompatible intent

Explanation: Reason code 44200 is received when opening the file. If you are trying to read, the file is already open for write. If you are trying to write, the file is already open.

User Response: Use *fanin* or *faninany*, as appropriate to your application, to merge the two streams; then use one *disk* stage to write the file.

System Action: The stage terminates with return code 746.

747E Not authorised to read *file*

Explanation: Reason code 44000 is received when opening the file for read. It could also be that the file was removed after *CMS Pipelines* determined that it exists, but this is probably unlikely.

System Action: The stage terminates with return code 747.

748E Disk mode is full

Explanation: Reason code 90131 is received when writing to the file.

System Action: The stage terminates with return code 748.

749E File *file* is on OS or DOS minidisk

Explanation: The file status byte contains 8.

User Response: Use *qsam* to read the file.

System Action: The stage terminates with return code 749.

750E Incorrect input block format

Explanation: *deblock* MONITOR has read a block that contains a length field of binary zeros, but the remainder of the block does not consist entirely of binary zeros.

752E No default file pool defined

Explanation: A SFS file is to be read from the default file pool, but no file pool is currently the default. Reason code 90590 was received when locating the file.

User Response: Specify a file pool explicitly or use the CMS command SET FILEPOOL to set the default file pool.

System Action: The stage terminates with return code 752.

753E NAMEDEF too long in *string*

Explanation: Reason code 90510 was received from DMSVALDT. A temporary name for a file name and type is longer than 16 characters.

System Action: The stage terminates with return code 753.

754E Improper use of stage; reason code *number*

Explanation: A built-in program that is reserved for IBM use has been invoked in a way that is not correct. The reason codes are:

- 1 Argument string is not eight bytes long.
- 2 The input record is not the length in the argument string.

System Action: The stage terminates with return code 754.

755E Offset not shorter than width

Explanation: The length of the offset specified (either as a number or as the length of the delimited string) is equal to or greater than the width.

System Action: The stage terminates with return code 755.

756W Use the := assignment operator instead of =

Explanation: A single equal sign is scanned.

User Response: Change to use the colon equal operator.

757W Use the ¬ operator instead of !

Explanation: An exclamation point is scanned.

User Response: Change to use the not operator.

758W Do not double up relational operators

Explanation: A double bar or a double ampersand is scanned.

User Response: Change to use a single operator character.

759E Incompatible types

Explanation: An operation is requested between a string and a counter. Relational operators must be between like types. Strings cannot be used with computational operators.

System Action: The stage terminates with return code 759.

760E No data will be available for input field

Explanation: An input range is specified after EOF without SELECT SECOND in effect. Thus, there are no data available to *specs* to supply.

User Response: Use SELECT SECOND to refer to the second reading station, where a copy of the last record is. However, if you were not using the second reading station and the field you require can be stored in a counter, it is more efficient to save the value in a counter while processing the detail record and then refer to the contents of this counter after the EOF item.

System Action: The stage terminates with return code 760.

761E Different key fields not allowed with AUTOADD

Explanation: AUTOADD was specified and the key field is defined in a different place in the detail and in the master records. This would make adding the record ambiguous.

User Response: Use *specs* to move the key field in the master or the detail records.

System Action: The stage terminates with return code 761.

762E Return code *number* reason code *number* from TSO

Explanation: The TSO command service routine (IKJEFTSR) gave the return code and reason code shown.

User Response: Refer to *TSO Programming Services*, SC28-1875.

System Action: The stage terminates with return code 762.

763E File token *word* is not valid (reason code *number*)

Explanation: The file token could not be converted from hexadecimal to binary. The reason codes are:

- 1 Leading blank in field.
- 2 Trailing blank in field.
- 3 Odd number of hex digits.
- 4 Digit not hex.
- 5 Output field exhausted.

Reason codes 1 and 2 should not occur.

System Action: The stage terminates with return code 763.

764E **Timestamp too short:***string*

Explanation: The time stamp must contain at least eight digits.

System Action: The stage terminates with return code 764.

765E **Timestamp too long:***string*

Explanation: The time stamp must contain at most fourteen digits.

System Action: The stage terminates with return code 765.

766E **Century incorrect in timestamp:** *string*

Explanation: The first two characters of the timestamp are less than 19.

System Action: The stage terminates with return code 766.

767E **Not numeric character in timestamp:** *string*

Explanation: A character of the time stamp is not numeric.

User Response: The timestamp is specified as a sequence of digits without the usual delimiter characters.

System Action: The stage terminates with return code 767.

768E **Incorrect record in file; reading record** *number*

Explanation: Return code 8 reason code 90117 is received on DMSREAD. This indicates that the file contains an impossible record length; it could be larger than the record length of the file or it could be zero.

User Response: Contact your systems support staff to have the problem diagnosed.

System Programmer Response: Open the file using DMSOPDBK and then read the blocks of the file with *filetoken* BLOCKED. Pass this file to *deblock* CMS to validate the file. If *deblock* does not issue a message, the record length is zero, indicating a premature end-of-file.

System Action: The stage terminates with return code 768.

769E **SYSOUT Class** *char* **is not a letter**

Explanation: A single character is specified, which is neither an asterisk, a letter, nor a digit. Or the keyword CLASS is specified and not followed by a one-character operand.

User Response: Use the keyword OUTDESC to specify a one-character output descriptor.

System Action: The stage terminates with return code 769.

770E **Period missing in destination** *word*

Explanation: A DESTINATION keyword is met, but the following word contains no period.

User Response: Make sure the destination contains both a system ID (also known as a node ID) and a user ID:

```
| sysout dest dkibvm2.john
```

System Action: The stage terminates with return code 770.

771E **Leading period in destination** *word*

Explanation: The first character of the destination is a period. This implies a null node ID.

System Action: The stage terminates with return code 771.

772E **Ending period in destination** *word*

Explanation: The last character of the destination is a period. This implies a null user ID.

System Action: The stage terminates with return code 772.

773E **Node** *word* **is not defined to JES**

Explanation: The first component of the destination is not known to JES.

System Action: The stage terminates with return code 773.

774E **Syntax error:** *explanation*

Explanation: REXX signalled a syntax error. The error text is substituted.

System Action: The stage terminates with return code 774.

775E **Incorrect file name** *word*

Explanation: Reason code 90420 is returned by DMSVALDT. The file name is longer than eight characters or contains a character that is not allowed.

System Action: The stage terminates with return code 775.

776E **Incorrect file type** *word*

Explanation: Reason code 90430 is returned by DMSVALDT. The file type is longer than eight characters or contains a character that is not allowed.

System Action: The stage terminates with return code 776.

777E **Incorrect file mode number** *word*

Explanation: Reason code 90430 is returned by DMSVALDT. The file mode number is not a digit between "0" and "6".

System Action: The stage terminates with return code 777.

778E Forbidden character in file name or file type *words*

Explanation: Reason code 90450 is returned by DMSVALDT. The file name or the file type contains an asterisk or a percent sign.

System Action: The stage terminates with return code 778.

779E Incorrect directory *word*

Explanation: Reason code 90430 is returned by DMSVALDT. The directory is longer than some limit or it contains a character that is not allowed.

System Action: The stage terminates with return code 779.

780E You are not allowed to write to file**Explanation:**

- The directory record for an existing file indicates that you cannot write to it. The reason can be that you do not have write authorisation or that the file is in a DIRCTL directory, which is currently accessed for write by some other user.
- Reason code 44000 is received when opening the file for append or replace. It is a remote possibility that the file was removed between the time *CMS Pipelines* determined that the file existed and the time it was opened.

System Action: The stage terminates with return code 780.

781E Incorrect file token *hex*

Explanation: The file token parameter does not refer to an open file.

User Response: If you have opened the file in a REXX program, remember to convert the token to printable hexadecimal before using it with the *filetoken* built-in program:

```
call csl 'dmsopen ... filetoken ...'  
'pipe filetoken' c2x(filetoken) '|...'
```

System Action: The stage terminates with return code 781.

782E Open intent is incompatible with stage position
(intent is *char*)

Explanation: The file token parameter identifies a file that is open with an intent that is not compatible with the position of the *filetoken* stage in the pipeline. A read intent is required when it is first in a pipeline; a write or replace intent is required when it is not first in a pipeline.

System Action: The stage terminates with return code 782.

783E Storage group space limit exceeded

Explanation: The storage group is full. It is not possible to write more data into the storage group. You may or may not have exceeded your space quota for the storage group.

System Action: The unit of work is rolled back. The stage terminates with return code 783.

784E Space quota exceeded

Explanation: You have exceeded your space quota for the storage group, or you have no space quota.

System Action: If the file was opened successfully, the unit of work is rolled back. The stage terminates with return code 784.

785E DMSOPBLK is not supported

Explanation: The file token represents a file that is opened through the DMSOPBLK callable service.

User Response: Use DMSOPDBK instead.

System Action: The stage terminates with return code 785.

786E Specified work unit does not exist

Explanation: Reason code 90540 is received from the callable service DMSEXIST.

System Action: The stage terminates with return code 786.

787E Too much ESM data (*number bytes*)

Explanation: More than eighty characters are specified for ESM data. This is over the maximum allowed by SFS.

System Action: The stage terminates with return code 787.

788E File pool is not available

Explanation: Reason code 97500 was received from DMSEXIST. The specified file pool or the one set by SET FILEPOOL is not known to CP.

User Response: Check your spelling carefully. Note that SET FILEPOOL does not report an error if the specified file pool is not known to CP.

System Action: The stage terminates with return code 788.

789E SAFE can be specified only for PRIVATE work unit

Explanation: SAFE was specified, but the stage is not using a private unit of work.

System Action: The stage terminates with return code 789.

790E File locked by other user or other unit of work

Explanation: Reason code 2200 was received from DMSOPEN or DMSOPDBK. The specified file is locked by some other user or another of your units of work.

User Response: Be careful about using the WORKUNIT DEFAULT option with stages that would otherwise acquire a private unit of work. When you do so, *CMS Pipelines* cannot commit the default unit of work; you must do so yourself. (For example by this REXX instruction

```
call csl 'dmscomm c_rc c_reason'
```

If the a file is updated on the default unit of work and then opened for modification on a different unit of work, a locking conflict is evident to the SFS server, even though this may not be obvious to you.

System Action: The stage terminates with return code 790.

791E File was committed by other user or other unit of work

Explanation: Reason code 20000 was received from DMSCOMMT. You were creating a file at the same time as another user or another unit of work was creating the same object. The another user or another unit of work managed to commit the file first.

User Response: Create a null file and then replace it if it takes an appreciable amount of time to create the file, particularly if the file is generated as a result of asynchronous events. This ensures that you can gain exclusive access to the file.

System Action: The unit of work was rolled back by CMS. The stage terminates with return code 791.

792E Fitting placement incompatible with RPL

Explanation: A fitting Request Parameter List that references the stage specifies an initial operation (read or write) that is incompatible with the placement of the *fitting* stage.

System Action: The stage terminates with return code 792.

793E Initial RPL state is not valid: *number*

Explanation: A fitting Request Parameter List that references the stage specifies an initial state that is neither IDLE, READ, nor WRITE.

System Action: The stage terminates with return code 793.

794E More than one RPL refers to stage

Explanation: Two fitting Request Parameter Lists reference the stage. This is an error, because the stage only supports one request at a time.

System Action: The stage terminates with return code 794.

796E 370 accommodation must be turned on (CP SET 370ACCOM ON)

Explanation: A device driver that performs I/O to a device that is not supported for Diagnose A8 has received an operation exception on a 370-mode I/O instruction.

User Response: Unless you run applications that require 370ACCOM to be off to work correctly, you should turn this option on in your PROFILE EXEC by the CP command “set 370accom on”. If you cannot run with the 370 accommodation on permanently, turn it on before issuing the PIPE command and turn it off after the pipeline has completed.

System Action: The stage terminates with return code 796.

797E Program check code '*hex*'x on TIO to communications device

Explanation: An unexpected program check is received while testing if the communications device can be used.

User Response: Contact your systems support staff.

System Programmer Response: This may be an error in *CMS Pipelines*. The hexadecimal value substituted shows the program exception encountered. Have this code ready when reporting the error to IBM.

System Action: The stage terminates with return code 797.

798I Forcing pipeline stall

Explanation: The fitting interface ran its pipeline without any change to the *fitting* stage(s). A stall is forced, because no further action is possible and the pipeline would never complete.

System Action: The pipeline is stalled.

1000E Secondary vector too short for *epname* or entry not present; install current CMS Pipelines

Explanation: A filter requires a routine that is reached via the extension to the secondary entry point vector, but the actual vector found is either too short or does not contain the address of the specified entry point.

User Response: Issue “pipe query level” to determine which version of *CMS Pipelines* you are using. You need PIPELINE MODULE level 110C0004 for the first extension to the secondary vector; you may need a higher level for the entry point being tested.

System Action: The stage terminates with return code 1000.

1010E VMCF CVT not found

Explanation: There is no active VMCF address space.

User Response: Ensure that TCP/IP is installed correctly and that the IUCV started task is active.

System Action: The stage terminates with return code 1010.

1011E Return/condition code number on IUCV QUERY

Explanation: The query function fails with the condition code (CMS) or return code (z/OS) shown.

User Response: Refer to the description of the condition codes associated with the IUCV instructions in *CP Programming Services*, SC24-6272.

System Action: The stage terminates with return code 1011.

1012E Return/condition code number on IUCV declare buffer

Explanation: *CMS Pipelines* is unable to declare the IUCV buffer. The condition code (CMS) or return code (z/OS) is substituted.

User Response: Refer to the description of the IUCV instructions in *CP Programming Services*, SC24-6272.

System Action: The stage terminates with return code 1012.

1013E No IUCV paths can be connected

Explanation: The maximum number of paths returned by IUCV QUERY is zero.

System Action: The stage terminates with return code 1013.

1014E IPAUDIT hex

Explanation: The audit field is not all zero bits.

User Response: Refer to the description of the IUCV SEND instruction in *CP Programming Services*, SC24-6272.

System Action: Message 1022 is issued to display the parameter list. The stage terminates with return code 1014.

1015E ERRNO number: chars

Explanation: The TCP/IP sets the error number shown.

System Action: The stage terminates with return code 1015.

User Response: The error number is defined for the socket calls described in *TCP/IP Version 2 for MVS: Programmer's Reference*, SC31-6087; refer to `bind()`, `select()`, `sendto()`, `socket()`, and `recvfrom()`.

1016I Reason: chars:

Explanation: The reason given by the server for severing the path.

1017E Unable to connect to server

Explanation: A nonzero return code was received when connecting to the service.

System Action: Messages 1018 are issued to display the parameter list. The stage terminates with return code 1017.

User Response: Ensure that TCP/IP is started and ready to process socket calls.

1018I ... hex: hex char

Explanation: Three lines are displayed for the IUCV parameter list and the ECB that are used for the request. Each line contains the hexadecimal storage address of the beginning of the data displayed. Up to 16 bytes are displayed in unpacked hexadecimal with character equivalents in EBCDIC.

1019E Input record is shorter than 24 bytes (it is number)

Explanation: An input record is too short to contain the complete network address prefix. The prefix contains these fields:

- Four bytes containing the timeout value in seconds.
- Four bytes of flags which are usually binary zeros.
- A halfword binary constant of 2. (Meaning that this is an Internet address.)
- A halfword containing the destination port number.
- A fullword containing the network address of the network interface where the destination port is.
- Eight bytes of binary zero.

System Action: The stage terminates with return code 1019.

1020E Socket operation cancelled (message is purged)

Explanation: *udp* finds its ECB posted with the code used by *pipestop* to indicate that the stage should terminate.

System Action: The stage terminates with return code 1020.

1021I Path number is connected for application

Explanation: Informational message that a connection complete interrupt has been fielded. This message is issued when the bit for 16 is on in the message level.

1022I IPARML: message (R0=number)

Explanation: Trace message issued when the bits for 128 or 64 are on in the message level. The message further describes the operation being traced. The number is decoded when it represents a valid IUCV code.

1023E All application slots in use

Explanation: There are too many applications in use for TCP/IP. As many slots are allocated as there can be IUCV connections. On z/OS the maximum number of paths is usually 255. One slot (and path) is allocated by each invocation of the *udp* built-in program.

System Action: The stage terminates with return code 1023.

User Response: Consider serialising the function being performed.

1032E Not a valid field identifier: *word*

Explanation: One letter is required for the field identifier. The word is longer than one character or it does not contain one of the characters from a to z (either case).

System Action: The stage terminates with return code 1032.

1033E Field ID is not defined

Explanation: The field being referenced has not been declared.

System Action: The stage terminates with return code 1033.

1036E Field ID is already defined

Explanation: The field identifier has been declared for a previous specification item.

System Action: The stage terminates with return code 1036.

1037E Field identifiers cannot be defined in break items

Explanation: A field identifier is specified after a BREAK item has been scanned. Specification items for breaks cannot have field identifiers.

System Action: The stage terminates with return code 1037.

1038E Not a decimal number: "*word*"

Explanation: A field to be loaded into a counter does not contain a valid decimal number or the number is too large.

System Action: The stage terminates with return code 1038.

1039E Counter overflow

Explanation: The exponent of a counter has overflowed.

Amazing! (Unless you did this on purpose.) The counter can store numbers of the order $10^{2000000000}$, whereas there are about 10^{80} atoms in the visible universe. What were you counting?

System Action: The stage terminates with return code 1039.

1040E Hex data too long (*number bytes*)

Explanation: A counter was loaded from a packed field that has more significant digits than the counter can contain.

System Action: The stage terminates with return code 1040.

1041E Multiplication overflow

Explanation: One of the two numbers being multiplied contains too many digits.

System Action: The stage terminates with return code 1041.

1048E Data not packed decimal: X'*hex*'

Explanation: A counter is to be updated, but the operand is not a valid packed field.

System Action: The stage terminates with return code 1048.

1049E More than one decimal point in data: *string*

Explanation: A string is converted to packed decimal. The string contains two decimal points.

System Action: The stage terminates with return code 1049.

1050E Counter contains more digits than picture: *string*

Explanation: The contents of the counter cannot be formatted with the picture specified.

System Action: The stage terminates with return code 1050.

1074E Too many nested IFs

Explanation: More than fifteen nested IFs are met.

User Response: Simplify the structure; try to use the conditional operator instead of IF.

System Action: The stage terminates with return code 1074.

1075E THEN expected; word was found

Explanation: A condition expression has been scanned after IF or ELSEIF, but there is no further data or the next word is not THEN.

System Action: The stage terminates with return code 1075.

1076E Unexpected keyword: word

Explanation: A ELSEIF, ELSE, or ENDIF is met, but there is no IF or ELSEIF to match it with.

System Action: The stage terminates with return code 1076.

1077E ENDIF expected; word was found

Explanation: A condition expression has been scanned after ELSE. The IF statement cannot have further condition clauses.

System Action: The stage terminates with return code 1077.

1078E Function does not support arguments; word was found

Explanation: A function reference was found for a function that does not accept arguments.

User Response: Write an empty parameter list: `first()`.

System Action: The stage terminates with return code 1078.

1079E Function requires one-character argument; "word" was found

Explanation: A function reference was found. The argument was empty or too long. The argument string is substituted.

System Action: The stage terminates with return code 1079.

1080E Incomplete IF

Explanation: The list contains more IF than ENDIF items.

System Action: The stage terminates with return code 1080.

1081E Unexpected character char

Explanation: The character shown is not valid at the point where it is.

System Action: The stage terminates with return code 1081.

1082E Missing colon

Explanation: Two question marks are met without an intervening colon

User Response: Enclose the innermost conditional in parentheses if you wish to perform a conditional within the "true" branch of an outer conditional.

System Action: The stage terminates with return code 1082.

1083E Assignment is not to a counter

Explanation: An equal sign or an update operator is met, but the left hand side is not a counter.

User Response: Use two equal signs to test two terms for equality.

System Action: The stage terminates with return code 1083.

1084E BREAK items are not allowed after EOF item

Explanation: BREAK, NOBREAK, or a second EOF is met after EOF has been specified.

User Response: Use IF to test for end-of-file rather than EOF if you do wish subsequent specification items to be executed for detail records.

System Action: The stage terminates with return code 1084.

1085E Counter number expected

Explanation: A number sign (#) is met indicating a counter, but the next character is not a digit.

System Action: The stage terminates with return code 1085.

1086E Improper operand for string expression

Explanation: A strictly compare operator or a colon for selection of expressions is met, but its operands are not strings, references to input fields, or the result of a conditional operator with string operands.

System Action: The stage terminates with return code 1086.

1087E String operand not acceptable to operator

Explanation: An operator is met that requires numeric operands, but one of its operands is a literal string.

System Action: The stage terminates with return code 1087.

1088W Last operation is not assignment

Explanation: SET is specified to compute an expression, but the result is not stored. Since it is discarded by *CMS Pipelines*, this is unlikely what you had in mind.

User Response: Remember that the assignment operator is colon equal (:=). An equal sign is the comparison operator.

System Action: Processing continues.

1089E Too many counters

Explanation: The span of numbers used for counters is so large that it would require more than a 2G area to store them all.

User Response: Try to reduce the span of counter numbers in use.

System Action: The stage terminates with return code 1089.

1090E Unrecognised operator word

Explanation: One or more operator characters are met, but the aggregate string is not a valid operator.

System Action: The stage terminates with return code 1090.

1091E Operator expected; found word

Explanation: An operator or the end of the expression is expected, but the character shown was met.

User Response: *specs* does not support the comma operator. Use the SET and semicolon operators instead.

System Action: The stage terminates with return code 1091.

1100E Record descriptor is too small (it contains number)

Explanation: A TCP/IP device driver that is specified with SF or SF4 detects a record that contains a record descriptor that is shorter than its own length.

User Response: Ensure that the application sending data to you observes the protocol you expect. In particular, pay attention to word sizes and byte ordering.

System Action: The stage terminates with return code 1100.

1110I Received number bytes

Explanation: The bit for 16 is on in the message level. A data packet is received. Zero bytes means end-of-file.

1111I Sent number bytes

Explanation: The bit for 16 is on in the message level. A data packet is sent.

1112I Closing socket (reason number)

Explanation: The bit for 16 is on in the message level. The socket is being closed.

1113I Purging IUCV message

Explanation: The bit for 32 is on in the message level. An IUCV operation is purged because an input record has arrived.

1114I IUCV reply number bytes

Explanation: The bit for 32 is on in the message level. An IUCV reply was received.

1115I Socket call for type

Explanation: The bit for 32 is on in the message level. A socket function is passed to TCP/IP.

1120E Stage cannot run in CMS subset

Explanation: The stage requires an interface that is not supported in CMS subset mode.

User Response: Issue the RETURN command to return to full CMS.

System Action: The stage terminates with return code 1120.

1121E Stage cannot run while DOS is ON

Explanation: The stage requires an interface that is not supported in CMS DOS mode.

User Response: Issue the SET DOS OFF command.

System Action: The stage terminates with return code 1121.

1122E Expression result is a string: string

Explanation: The result of an expression is a character string rather than a number in a context where a string is not valid.

User Response: Inspect the expressions in IF, PRINT, and SET clauses for single literals, such as these:

```
. ... | spec set "sh" | ...
. ... | spec ... if #0?'a':"b" then ... | ...
. ... | spec print "sh" picture 99 1 | ...
```

System Action: The stage terminates with return code 1122.

1123E Unacceptable input record length *number*

Explanation: An input record that is not null is not the required length. For *socka2ip*, the input record is neither four nor sixteen bytes.

System Action: The stage terminates with return code 1123.

1124E Incorrect NAMEDEF *word (a directory name must contain a period)*

Explanation: Reason code 90530 is returned by DMSVALDT. The third word of the argument string is taken as a name definition for the directory containing the file, but there is no such name defined.

User Response: You probably wanted to refer to a directory. Ensure that the third word contains either an explicit file pool name and user ID, or at least one period. For example, to read from JOHN's top level directory in the current file pool:

```
pipe < profile exec john. | hole
```

System Action: The stage terminates with return code 1124.

1125E No space left in PDS directory

Explanation: Return code 12 reason code 0 is received on the STOW macro. This indicates that the directory is full.

System Action: The stage terminates with return code 1125.

1126E Record descriptor indicates *number bytes, but minimum is number*

Explanation: When deblocking records that contain their record length (such as SF4), the record descriptor indicates a record length that would not include the record descriptor itself. For example, an input record that contains four binary zeros is not valid for *deblock* SF4.

System Action: The stage terminates with return code 1126.

1127E Host name too long: *string*

Explanation: A word of an input record to *nsquery* is longer than 1024 characters. If the word contains no periods, the length of the word plus the length of the domain origin (the argument to *nsquery*) is larger than 1023. This limit is imposed by the domain name system.

System Action: The stage terminates with return code 1127.

1128E Two consecutive periods in host name: *string*

Explanation: A word of an input record to *nsquery* contains two adjacent periods, which would indicate a null component in the host name. If the word contains no periods, the domain origin (the argument to *nsquery*) contains a leading period or two consecutive periods.

System Action: The stage terminates with return code 1128.

1129E Component of host name too long: *string*

Explanation: A component of a host name (or of the domain name, which is the argument to *nsquery*) Contains more than 255 consecutive characters without a period. This limit is imposed by the domain name system.

System Action: The stage terminates with return code 1129.

1130E Variable *name is not defined in file* *string*

Explanation: A variable is not defined in the TCPIP DATA file.

System Action: The stage terminates with return code 1130.

1131E Name server on port *number at IPaddress* **timed out**

Explanation: No response is received from the name server after the number of retries specified in TCPIP DATA.

System Action: The stage terminates with return code 1131.

1132E Name server response is truncated

Explanation: The response from the name server indicates that the response was truncated.

System Action: The stage terminates with return code 1132.

1133E Name server query in wrong format

Explanation: The name server return code is 1.

System Action: The stage terminates with return code 1133.

1134E Host *word does not exist*

Explanation: The name server return code is 3 and the secondary output stream to *nsresponse* is not defined.

System Action: The stage terminates with return code 1134.

1135E • 1145W

1135E Host *word* does not exist

Explanation: The name server return code is 0, but no response is returned. The secondary output stream to *nsresponse* is not defined. The word is recognised as a domain, not as a host.

System Action: The stage terminates with return code 1135.

1136E Return code from name server: *number*

Explanation: The return code substituted was returned by the name server.

User Response: Refer to the current RFC for the meaning of the return code. This RFC is likely to have replaced RFC 1035.

System Action: The stage terminates with return code 1136.

1137E Unexpected response record type: *number*

Explanation: The first response record from the name server is not an address record.

User Response: Refer to the current RFC for the meaning of the record type. This RFC is likely to have replaced RFC 1035.

System Action: The stage terminates with return code 1137.

1138E Unsupported RESOLVEVIA: *word*

Explanation: The method specified in the TCPIP DATA file for name resolution is not supported.

System Action: The stage terminates with return code 1138.

1139I Query summary state of streams

Explanation: Pipeline dispatcher trace is active. The stage requests the summary status of its streams.

System Action: None.

1140E Unable to resolve *word* (RXSOCKET is not available)

Explanation: The host name could not be resolved because the RXSOCKET interface was not available.

System Action: The stage terminates with return code 1140.

1141E Unable to resolve *word* (RXSOCKET did not return a result)

Explanation: The host name could not be resolved because the RXSOCKET interface did not return a result on the function invocation.

System Action: The stage terminates with return code 1141.

1142E Unable to resolve *word* (RXSOCKET error *string*)

Explanation: The host name could not be resolved because the RXSOCKET interface gave a return code.

System Action: The stage terminates with return code 1142.

1143E Unable to resolve *word* (RXSOCKET Version 2 is required)

Explanation: The host name could not be resolved because the RXSOCKET interface was downlevel. The string '-1' was returned; this is the way Version 1 reacts to errors.

System Action: The stage terminates with return code 1143.

User Response: Install RXSOCKET Version 2 or VM/ESA Version 2 Release 2, which has RXSOCKET built in. Name resolution will work with Version 1 if you initialise the socket interface externally to *CMS Pipelines*.

1144E Key/ID field is not anchored at the extremities of the input record (*number* before; *number* after)

Explanation: The STRIP option was specified to delete the key field or the stream identifier from the output record. This is not possible since the field is inside the record. The number of bytes before the key field and the number of bytes after the key field are substituted.

System Action: The stage terminates with return code 1144.

1145W PIPE command was issued from XEDIT, which truncates at or before 255 characters (use Address Command in XEDIT macros)

Explanation: The PIPE command which caused an error message to be issued in the scanner was issued from XEDIT.

User Response: When issuing PIPE commands from XEDIT macros, be sure to remember to address them to COMMAND, rather than to the default XEDIT command environment. XEDIT truncates commands after 255 bytes without issuing a warning message.

1146E Expect OF; found: word

Explanation: The SUBSTR keyword was specified, but the closing OF was not found where it was expected. The word found is substituted. If the message ends in the colon, the the stage's argument string was too short.

System Action: The stage terminates with return code 1146.

1147E Creation time cannot be changed for an existing file

Explanation: Reason code 51051 was received from DMSCLOSE or DMSCLDBK

The file to be replaced exists and CMS refuses to change its creation date.

User Response: If you really wish to change the creation date for a file, rename the file and then use *CMS Pipelines* to create it again. You will effectively lose all authorisations that you may have granted on the file because they will stay with the renamed file.

System Action: The stage terminates with return code 1147.

1148E Expected parameter token "sysv"; found "word"

Explanation: The PIPMOD nucleus extension is called with a parameter token, but the first parameter token is not the one for the system services vector.

System Action: The stage terminates with return code 1148.

1149E Too many parameter tokens found (second is "word")

Explanation: The PIPMOD nucleus extension is called with a parameter token and the first parameter token is the one for the system services vector, but it is not followed by a fence indicating the end of the list of parameter tokens.

System Action: The stage terminates with return code 1149.

1150E Lost race for SCBWKWRD

Explanation: PIPMOD was invoked to initialise *CMS Pipelines*. When the initialisation started, the user word for the PIPMOD nucleus extension was zero, but by the time it came to set the user word, the SCBLOCK already had a nonzero user word.

System Action: The stage terminates with return code 1150.

System Programmer Response: Investigate whether the virtual machine runs a vendor multitasking system. If it

does, ensure that *CMS Pipelines* is initialised before the vendor multitasking package takes over control of CMS.

1161E Unable to load module name (return code number)

Explanation: The *hlasm* interface cannot load the interface module to invoke it on CMS. The LOADMOD command fails with the return code shown.

User Response: Ensure that Release 2 of the High Level Assembler product is installed; *CMS Pipelines* does not support release 1.

System Action: The stage terminates with return code 1161.

1162E Unable to find module name

Explanation: The *hlasm* interface cannot find the interface module in storage after it has been loaded. This is likely to be the result of a programming error in *CMS Pipelines* or a change in the response to the PROGMAP command. *CMS Pipelines* assumes that the response contains two lines and that the load address of the module is the second word of the second line; it further assumes that this word is in printable hexadecimal.

System Action: The stage terminates with return code 1162.

1163E Unable to declare exit

Explanation: The *hlasm* interface cannot declare its exit because an instance is already declared at some other address. The return code on the IDENTIFY macro instruction is X'14'.

User Response: Return to the CMS ready prompt to clear this stale exit pointer. Then retry the pipeline.

System Action: The stage terminates with return code 1163.

1164W Variable name is not valid: contents

Explanation: A pipeline global variable does not contain an acceptable value.

System Action: The value is ignored and the default is used instead.

1165E Configuration variable name is not recognised

Explanation: The configuration variable is not known to *CMS Pipelines*.

User Response: Note that the names of configuration variables must be spelt out. Some are longer than eight characters; they must be specified in their entirety. Case is ignored in the names of configuration variables.

System Action: The stage terminates with return code 1165.

1166E Keyword *name* is not recognised for configuration variable *name*

Explanation: The first word of an input line to *configure* is recognised as the name of a keyword configuration variable, but the keyword specified is not valid for that configuration variable.

User Response: Note that the keywords must be spelt out. Some are longer than eight characters; they must be specified in their entirety. Case is ignored in keywords.

System Action: The stage terminates with return code 1166.

1167I Cannot load message repository *word*

Explanation: *CMS Pipelines* was unable to add its message repository to the current language. Other CMS messages may also have been issued.

CMS Pipelines issued this command, where *xxx* contains the repository set for *CMS Pipelines* or inferred from its default style:

```
set language ( add xxx user
```

The default message repository is FPL.

User Response: Either make the message repository available, set the language to a language for which there is a message repository, or disable the repository by this command:

```
PIPE literal REPOSITORY -|configure
```

You will receive these nuisance messages until you take some action to avoid them.

System Action: Processing continues. The message was issued from the internal message table, which is in English.

1168E Cannot convert relative date format *word* to absolute date format *word*

Explanation: Some date formats are absolute; that is, they reference a particular moment in time. Other date formats are relative; that is they specify some amount of time. A relative date cannot be converted to an absolute date.

User Response: Change the input date format to an absolute date format or change the output date format to a relative date format. The default date format is an absolute date format. Therefore, when converting a relative date format you must specify a relative output date format.

System Action: The stage terminates with return code 1168.

1169E Variable *word* is not a token set by SCANRANGE (reason code *number*)

Explanation: The contents of the variable were not set by the *SCANRANGE* pipeline command. In particular, the variable may have no value. The reason codes are:

1. The variable is not set.
2. The variable contains too many characters.
3. The variable contains too few characters.
4. The check word in the variable is incorrect.

User Response: Be sure that you quote the name of the variable; it must not be substituted by REXX.

```
'scanrange required range1 .' arg(1)
'peekto line'

```

System Action: The stage terminates with return code 1169.

1170E The input date is not valid: *word* (reason code *number*)

Explanation: The date could not be converted. The reason code is from the *DateTimeSubtract* callable services library routine.

User Response: Correct the error described by the reason code for *DateTimeSubtract*. If the reason is not obvious, refer to *For Timer Services* section in the *Return and Reason Code Values* appendix of the *CMS Application Multitasking* manual to find the symbolic name for the reason code. (The VM library contains no further description of these error codes.)

System Action: The stage terminates with return code 1170.

1171W No output date format specified; the default output date format is the same as the input date format

Explanation: The output date format was not specified. It defaults to the *ISODATE* date format. This value is the same as the input date format.

System Action: Processing continues. The stage validates all dates as requested.

User Response: Specify the output date format to avoid this message when you wish to validate dates without converting them.

1172I Restoring fitting name *word*

Explanation: A *fitting* stage is terminating. The fitting name is restored in the Request Parameter List.

1173I No RPL to restore

Explanation: A *fitting* stage is terminating. The fitting name was not resolved and thus it cannot be restored.

1174E Not a hexadecimal address word

Explanation: The word is not the unpacked hexadecimal representation of a machine address. This may be caused by a corrupt control block structure for *CMS Pipelines*.

User Response: Do not invoke *fplricdf*; use *pipdump* or *jeremy* instead.

System Action: The stage terminates with return code 1174.

1175E Incorrect check word in PIPEBLOK: word

Explanation: This may be caused by a corrupt control block structure for *CMS Pipelines*. The check word is shown in unpacked hexadecimal. It should have been “pipe”, which is X'97899785'.

System Action: The stage terminates with return code 1175.

1176W The pointer to the Contents Vector table is destroyed (reason code number); investigate VM61261

Explanation: The pointer in low storage to the Contents Vector Table is destroyed. The reason codes are:

- 1 The CVT pointer is either zero or negative.
- 2 The CVT pointer does not point within the virtual machine.
- 3 The byte at offset X'74' in the CVT has the bit for X'40' zero. This bit (originally meaning the Primary Control Program) should be on for CMS.
- 4 The byte at offset X'74' in the CVT has one or more of the bits for X'3F' nonzero.

User Response: Contact your systems support staff.

System Programmer Response: Investigate whether corrective service is available. In particular, ensure that the fix for APAR VM61261 is applied.

Issue the command `cp trace store into 10.4` to set a trap for storing into the primary CVT pointer.

1177E The system does not support date format word

Explanation: The specified date format is not supported for the level of CMS on which you are running and in particular not by the version of the DMSDTS callable service you are using. The date format may be valid on a later level of CMS or it may not be valid for any level of CMS.

System Action: The stage terminates with return code 1177.

1178W The pointer to the Contents Vector table has been restored from the alternate pointer

Explanation: The pointer in low storage to the Contents Vector Table is destroyed, but the alternate pointer is intact.

User Response: Contact your systems support staff.

System Programmer Response: Investigate whether corrective service is available. In particular, ensure that the fix for APAR VM61261 is applied.

Issue the command `cp trace store into 10.4` to set a trap for storing into the primary CVT pointer.

1179W The alternate pointer to the Contents Vector table has been restored from the primary pointer

Explanation: The alternate pointer in low storage to the Contents Vector Table is destroyed, but the primary pointer is intact.

User Response: Contact your systems support staff.

System Programmer Response: Investigate whether corrective service is available. In particular, ensure that the fix for APAR VM61261 is applied.

Issue the command `cp trace store into 500.4` to set a trap for storing into the alternate CVT pointer.

1180E A directory in the path string does not exist or you are not authorised for it

Explanation: Reason code 44000 was received when creating a file. Reason code 90230 was received when opening a directory.

System Action: The stage terminates with return code 1180.

1181E Directory control directory string is accessed read only

Explanation: Reason code 63700 was received when opening the file.

System Action: The stage terminates with return code 1181.

1182E Date format word cannot be used as an input date format

Explanation: The specified date format can be used only as an output date format.

System Action: The stage terminates with return code 1182.

1183E • 1197E

1183E Date cannot be converted; input date *word* is not valid

Explanation: The input date (that is, the contents of the specified field in an input record) is not a valid date for the input date format specified. Possible causes for this error include:

- The input date format does not match the input date.
- The input range includes information other than the date.
- The input range is not the correct syntax for a date of the specified format.
- The input date specifies a date which cannot occur, such as, February 29, 1997 (1997 is not a leap year).

System Action: The stage terminates with return code 1183.

1184E Input date *word* cannot be expressed in the output date format

Explanation: The input date has no meaning for the output date format. This occurs when one of three conditions exists:

- The input date is prior to the epoch for the output date format. For example, it is before January first 1900 for TOD Absolute.
- The input date is negative and the output date format is MET.
- The input date specifies a negative year and the output date format is not SCIENTIFIC_ABSOLUTE.

System Action: The stage terminates with return code 1184.

1185E Cannot convert absolute date format *word* to relative date format *word*

Explanation: Some date formats are absolute; that is, they reference a particular moment in time. Other date formats are relative; that is they specify some amount of time. An absolute date cannot be converted to a relative date.

System Action: The stage terminates with return code 1185.

User Response: Change the input date format to a relative date format or change the output date format to an absolute date format.

1186W Operand *string* is ignored for input date format *word*

Explanation: You specified the use of a sliding window for an input date format of REXX_DATE_C or REXX_DATE_D.

System Action: The sliding window operand is ignored. The date is converted using a base year of the current century for REXX_DATE_C or the current year for REXX_DATE_D.

User Response: Do not specify a sliding window for an input date format of REXX_DATE_C or REXX_DATE_D.

1191I Close flags *string*

Explanation: The callable service DMSCLOSE gave an unexpected return code. The close flags are displayed.

1192I Close flags *string*; record length *number*, count *number*

Explanation: The callable service DMSCLDBK gave an unexpected return code. The close flags, record length, and record counts are displayed.

1193I Current input stream *number* has record available

Explanation: Tracing message. SELECT ANYINPUT is issued and the producer on the currently selected input stream has a record available.

1194I Producer on input stream *number* has record available

Explanation: Tracing message. SELECT ANYINPUT is issued. A producer other than the currently selected one has a record available.

1195I Selecting input stream *number*

Explanation: Tracing message. The stage is waiting after SELECT ANYINPUT has been issued. A record is now available.

1196E Do not connect unused input stream *stream*

Explanation: A stream is connected that the stage does not use. This is often a symptom of an incorrect placement of a label reference. *lookup* detects that input stream 4 or 5 is connected.

User Response: Verify the streams to the stage in question. In particular, pay attention to streams that are used both for input and output; there must be only one label reference to represent both the input and the output stream.

System Action: The stage terminates with return code 1196.

1197E Do not connect unused output stream *stream*

Explanation: A stream is connected that the stage does not use. This is often a symptom of an incorrect placement of a label reference.

User Response: Verify the streams to the stage in question. In particular, pay attention to streams that are used both for input and output; there must be only one label reference to represent both the input and the output stream.

System Action: The stage terminates with return code 1197.

1198I Stage is active

Explanation: The ABEND recovery routine found an active stage. Additional messages are issued to identify the stage, if enabled in the message level for the stage.

1210I Request for *hex* doublewords unsuccessful (from *hex*)

Explanation: Free storage management trace is enabled. The storage request failed. Note that the number of doublewords are shown in hexadecimal.

1211I Got *hex* doublewords at *hex*

Explanation: Free storage management trace is enabled. A block of storage is allocated.

1212I Rel *hex* doublewords at *hex*

Explanation: Free storage management trace is enabled. A block of storage is deallocated.

1213E Storage at *address* is not on allocated chain

Explanation: Free storage management validation is enabled. The block being released is not on the chain of storage that was allocated by *CMS Pipelines*.

System Action: The storage is not returned to the operating system. Processing continues.

1214E Storage at *address*; check word at *address* is destroyed

Explanation: Free storage management validation is enabled and a block of storage is being released. The fullword immediately after the allocated area has been corrupted.

System Action: The storage is not returned to the operating system. Processing continues.

User Response: It is likely that the storage allocation is insufficient.

1215E Storage at *address* allocated *hex* doublewords; releasing *hex*

Explanation: Free storage management validation is enabled and a block of storage is being released. The size being returned is not the one allocated.

System Action: The storage is not returned to the operating system. Processing continues.

User Response: Be sure to return the number of doublewords actually allocated; this number may be larger than the number of doublewords requested.

1216E Storage at *address*; check word is destroyed

Explanation: Free storage management validation is enabled and a block of storage is being released. The fullword immediately before the allocated area has been corrupted.

System Action: The storage is not returned to the operating system. Processing continues.

User Response: It is likely that the error is an insufficient request for the storage that preceded this particular block of storage. This is a particularly nasty bug to track down, especially on TSO.

1217I Contents: *hex*

Explanation: Free storage management validation is enabled and an error is discovered when a block of storage is released. The first thirty-two bytes of the allocated block are displayed.

1220E IEANTRT RC=*hex* not equal to R15 (=hex.)

Explanation: The FPLDEBUG command found that the return code stored by IEANTRT is not equal to the return code in general register 15.

System Action: The command terminates with return code 1220.

1221I The TSO Pipelines name/token is not established

Explanation: IEANTRT gave return code 4. No PIPE command has been issued for the address space or FPLRESET has been issued to delete the *TSO Pipelines* global information area.

1222E IEANTRT RC=*hex*

Explanation: The FPLDEBUG command received a return code that is neither 0 nor 4.

System Action: The command terminates with return code 1222.

1223E Nametoken field *name* contains *value*; expect *value*

Explanation: The token returned by the IEANTRT callable service does not contain the expected data.

System Action: The command terminates with return code 1223.

1224I TSO Pipelines global area is at *hex*

Explanation: This is a debugging message.

1225E • 1232E

1225E ABEND *hex* accessing the global data area

Explanation: The FPLDEBUG command received a program check when it tried to verify the address of the global data area returned in the name/token.

System Action: The command terminates with return code 1225.

1226E Global area is corrupted

Explanation: The FPLDEBUG command could not verify the first twenty-four bytes of the global data area returned in the name/token.

System Action: The command terminates with return code 1226.

1227E Insufficient data returned by DMSGETDI; got *number* expect *number*

Explanation: The callable service returned fewer bytes than required for the function requested.

User Response: If FPLSTAT was specified and no output is produced, your system is does not support the function requested.

System Programmer Response: If FPLSTAT is omitted and some output was produced, the combination of buffer size used by *sfsdir* and the record length returned is such that a partial record is generated in a downlevel format at the end of the buffer. Open an APAR against DMSGETDI.

1228I Return code *number* erasing work file

Explanation: While replacing a file in a SFS directory that is accessed as a mode letter using *>m_{sk}*, the file is created correctly, but erasing the work file fails with the return code substituted. The configuration variable DISKREPLACE is set to COPY.

System Action: The error is ignored.

User Response: Do not panic. The file is created correctly; most likely the work file had mode number 3 and the copy operation has removed the work file. The return code is 28 in this case.

1229E Length code *char* is not valid

Explanation: The first position of an input record to *uu_{dec}raw* is not one of the valid characters for the encoding format.

User Response: Be sure to remove the header and trailer records from a file in the uuencode format.

System Action: The stage terminates with return code 1229.

1230E Expect "begin", found *string*

Explanation: The first input record to *uu_{dec}ode* does not contain the word "begin" followed by a blank.

System Action: The stage terminates with return code 1230.

1231E Expect "end", found *string*

Explanation: The input record to *uu_{dec}ode* that follows an encoded file does not contain the word "end".

System Action: The stage terminates with return code 1231.

1232E Native sockets are not available (reason code *number*)

Explanation: *CMS Pipelines* is unable to call the service routine to create a TCP/IP socket using the native z/OS callable service BPXISOC. The reason code indicates which particular test failed:

- 1 The CVT field CSRTABLE does not contain a pointer. It contains -1. (Offset 544, decimal).
- 2 The resource slot in the OpenExtensions callable services function table points to a dummy entry.
- 3 The OpenMVS slot in the CSR table does not contain a pointer. It contains zero. (Offset 24, decimal).
- 4 The table of entry points to OpenExtensions callable services is too short to contain the function requested. That is, the operating system does not support the function requested.
- 5 The table of entry points to OpenExtensions callable services contains a dummy entry for the function requested. That is, the operating system does not support the function requested or the function is not available in this particular configuration.
- 10 The CMS is a release where the simulated CVT is too short to contain the pointer to the CSRTABLE.
- 4 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed. It is either zero or negative.
- 5 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed. It does not point within the virtual machine.
- 6 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed, or the CVT has been corrupted. The byte at offset X'74' has the bit for X'40' zero. This bit indicates that the system is CMS.
- 7 The pointer to the Contents Vector Table (CVT) in location 16 (decimal) is destroyed, or the CVT has been corrupted. The byte at offset X'74' has one or more of the bits for X'3F' nonzero. These bits must be zero on CMS.

User Response: Contact your systems support staff if the reason code indicates that the CVT pointer is destroyed. These reason codes are issued on CMS only.

System Programmer Response: Investigate if corrective service is available. In particular, ensure that the fix for APAR VM61261 is applied.

System Action: The stage terminates with return code 1232.

1233E ERRNO *string* reason *string* in *string*

Explanation: An error has occurred in the native z/OS TCP/IP stack. The decoded error number and reason code are shown. If the numbers are not known, the actual numbers are substituted.

User Response: Refer to *OpenExtensions Programming: Assembler Callable Services Reference* or to the member BPXYERNO in the system macro library.

System Action: The stage terminates with return code 1233.

1234I Attention exit disabled. Hit attention to terminate command

Explanation: The attention routine was entered a third time. Further attentions will cause TSO to terminate the command or be handled by REXX.

1235I Reason Code *hex (hex) number (decimal)*

Explanation: The reason code for an error from callable services is shown. The first substitution displays the entire reason code in hexadecimal. The second displays the right-most halfword in decimal; this is the number defined with the JR reason codes in the macro BPXYERNO.

1236I USS return code *number* reason *hex* function: *word*

Explanation: A call to the Unix System Services failed, but the stage will recover or provide its own diagnostics. This message is issued when message level 1024 is set.

The return code is shown as a decoded errno if the value is one of those recognised by *CMS Pipelines*; otherwise the return code is shown as a decimal number.

The reason code is shown as a decoded value if the value is one of those recognised by *CMS Pipelines*; otherwise the reason code is shown in hexadecimal. Only the last four digits are significant.

The function is shown as the name of the equivalent POSIX function.

User Response: Refer to *OS/390 Messages and Codes* for the error codes.

System Action: The error is percolated.

1237I Active process and thread IDs:*hex (hexadecimal)*

Explanation: The FPLDEBUG command is issued on z/OS. The process and thread IDs are displayed in hexadecimal for all threads on which *TSO Pipelines* is active.

1238I Shutting down for write

Explanation: Tracing message.

The socket is being shut down for write. This should cause the partner to see end-of-file, but still be able to send a response.

1239I About to receive from socket

Explanation: Tracing message.

The stage will enter a blocking read from the socket.

1240I Unknown CP/CMS command: *string*

Explanation: The command specified after the system service name was not recognised by the system.

System Action: The error is ignored by *CMS Pipelines*.

User Response: But you may wish to investigate. A likely cause is that keyword operands for the *starsys* stage were entered after the name of the system service rather than before.

1241I Suppressed CP/CMS command: *string*

Explanation: The *starsys* stage was unable to connect to a system service and as a result it did not issue the command shown.

System Action: The original error is reported.

User Response: Ensure that the command shown is indeed a command you wish to have executed. It is possible that the connect failed because the IPUSER field was not set correctly. Remember that keyword operands for the *starsys* stage must precede the name of the system service; the string after the name of the system service is a CMS command that is issued after the stage has connected to the system service.

1242I STOPECB *hex* called

Explanation: Tracing message. The ECB address is substituted, or "off".

1243T PSTV at *address* corrupt: *string*

Explanation: General register 9 has been destroyed.

1244E Input record contains incorrect data for ASCII quoted-printable format: X'*hex*'

Explanation: The character X'3D' is met in a position other than the last of an input record. Such an equal sign should be followed by two characters representing the hexadecimal encoding of four bits, but either there is only one character following the equal sign (four hexadecimal digits substituted) or the two characters are not valid hexadecimal (six hexadecimal digits substituted).

1245E • 1255E

Note that the hexadecimal digits should be represented in ASCII. Thus, the numbers should be from X'30' to X'39' and the letters X'41' through X'45'.

User Response: Be sure that the input is correct ASCII. If you are processing a mail file that has been converted to EBCDIC, be sure to translate it back to ASCII before passing it to *qpdecode*.

To make a robust decoder, connect the secondary output stream from *qpdecode* and perform error recovery or ignore the error.

System Action: The stage terminates with return code 1244.

1245E You need pipeline version *hex* for this stage

Explanation: A stage in a filter package has determined that it runs on a version of *CMS Pipelines* that does not provide sufficient infrastructure for the stage to work.

User Response: Install the required level of *CMS Pipelines*.

System Action: The stage terminates with return code 1245.

1246E You need a more modern VM (interrupt code *number*)

Explanation: A program check was received on a STSI instruction. This indicates that your release of VM/ESA does not support the instruction. It is indeterminate whether the hardware has the store-system-information facility installed.

User Response: Contact your friendly IBM salesperson to order the required software. Also enquire whether your hardware has the store-system-information facility installed.

System Action: The stage terminates with return code 1246.

1247E Incorrect selector value specified (interrupt code *number*)

Explanation: A program check was received on a STSI instruction to obtain the requested piece of system information. The hardware and software combination is known to support the store-system-information facility; thus the operand is not a valid combination of selectors.

User Response: The selector is specified as a three digit number.

System Action: The stage terminates with return code 1247.

1249E Requested SYSIB information not available

Explanation: A nonzero condition code was received on a STSI instruction to return the actual configuration information.

User Response: At the time of writing, these are the valid selectors:

- . 111 Basic machine.
- . 121 Basic machine CPU
- . 122 Basic machine CPUS.
- . 221 Logical partition CPU.
- . 222 Logical partition CPUS.
- . 322 Virtual machine CPUS.

System Action: The stage terminates with return code 1249.

1250E Incorrect code point X'*hex*' in first byte

Explanation: *utf8* DECODE STRICT met a byte that contains an incorrect value.

System Action: The stage terminates with return code 1250.

1251E Incomplete UTF-8 multibyte character

Explanation: *utf8* DECODE STRICT met a byte that contains the first half of an encoded value, but there are no more bytes in the input record.

System Action: The stage terminates with return code 1251.

1252E Incorrect code point X'*hex*' in second byte

Explanation: *utf8* DECODE STRICT met a byte that correctly begins a two-byte sequence, but the second byte contains a value that is not valid.

System Action: The stage terminates with return code 1252.

1253E Address X'*hex*' before section base

Explanation: Columns 5-8 of the input record contains a value that is smaller than the value in the first record for that particular section.

System Action: The stage terminates with return code 1253.

1254E Incorrect device number *hex* (larger than FFFF)

Explanation: The device number does not contain all zero bits in the leftmost halfword.

System Action: The stage terminates with return code 1254.

1255E CP paging error on diagnose 210 device number *hex*

Explanation: CP has set the condition code indicating that it was unable to provide information about the device.

System Action: The stage terminates with return code 1255.

1256E Cylinder number *number* beyond disk capacity *number*

Explanation: The disk has only the number of cylinders shown.

System Action: The stage terminates with return code 1256.

1257E Track number *number* beyond cylinder capacity *number*

Explanation: The disk has only the number of tracks shown.

System Action: The stage terminates with return code 1257.

1258E Number of tracks *number* beyond remaining device capacity *number*

Explanation: You have specified a track count that is larger than the total number of tracks remaining on the device.

System Action: The stage terminates with return code 1258.

1259E Volume does not have label specified

System Action: The stage terminates with return code 1259.

1260E Ending cylinder (*number*) lower than the beginning one

System Action: The stage terminates with return code 1260.

1261E Input record too long (it is *number*)

System Action: The stage terminates with return code 1261.

1264E Incorrect home address X'*hex*'

Explanation: The first byte (the flag byte) is not binary zeros, the cylinder and head fields do not contain data within the device geometry.

System Action: The stage terminates with return code 1264.

1265E Incorrect record 0 count field X'*hex*'

Explanation: The cylinder and head fields of record 0 are not equal to the ones in the home address or record 0 does not have a key length of 0 and data length of 8.

System Action: The stage terminates with return code 1265.

1266E Missing end of track marker

Explanation: The the last eight bytes of the record do not contain all one bits.

System Action: The stage terminates with return code 1266.

1267E Device number *hex* is read only

Explanation: An operation is rejected with sense bytes that include the write inhibited sense bit.

System Action: The stage terminates with return code 1267.

1268E Too many records on track (*number*)

Explanation: The track contains more than 255 records, including record 0. While such a track could be constructed, it is not supported by the way *CMS Pipelines* writes the track.

System Action: The stage terminates with return code 1268.

1269E Spurious end of track marker

Explanation: The count area of a record contains all one bits, but there remains more than eight bytes in the record.

System Action: The stage terminates with return code 1269.

1270E Record zero missing

Explanation: The track contains no record zero. All modern disks must have a record zero.

System Action: The stage terminates with return code 1270.

1271W Using obsolete version of *word word*

Explanation: *CMS Pipelines* determined that a program or file that is EXECLOADED is not the current version, but EXECLOAD failed to load the latest version. Message DMSEXLA14E is also issued.

The most likely cause for this message is that a REXX filter has been updated while it is executing and a new invocation is requested in a pipeline specification being added to the pipeline set.

System Action: The version already in storage is used.

User Response: To defeat *CMS Pipelines*'s loading and unloading of programs, use the EXECLOAD command to load your REXX filters before running the pipeline that uses them.

1272E Unable to find DMSEXI

Explanation: *CMS Pipelines* is running on CMS 14 or later, but the command table does not contain the command EXEC. Further, the work area for *rexx* is above the 16M line. As a consequence, there is no way to invoke the interpreter in the current environment.

System Action: The stage terminates with return code 1272.

User Response: Contact your systems support staff. As a circumvention, reduce the size of the virtual machine to less than 16M.

System Programmer Response: Investigate whether corrective service is available.

1273E Count area incomplete (number bytes available)

Explanation: The track after the last record contains less than eight bytes of data and is thus neither a valid count area nor a valid end of track marker.

System Action: The stage terminates with return code 1273.

1274E Record incomplete (number bytes available; number bytes required)

Explanation: The track after the last record contains a count area, but the count area indicates a longer record than the actual data present.

System Action: The stage terminates with return code 1274.

1275I Processing cylinder number track number record number

Explanation: Informational message when an incorrect track format is detected.

1276E Buffer length is not valid (hex doublewords requested)

Explanation: A buffer is to be extended to a size that would be negative and also larger than 31-bit addressing allows.

User Response: If this message is issued by a built-in program, this is an error in *CMS Pipelines*.

System Action: The stage terminates with return code 1276.

1277E Record length (number) is not 8+keylength+datalength (number)

Explanation: The input record does not represent a valid CKD block.

System Action: The stage terminates with return code 1277.

1278E Track number is not specified in input record number

Explanation: The input contains one word, not the minimum two.

System Action: The stage terminates with return code 1278.

1279E No messages in queue, but interrupt received.

Explanation: *iucvclient* or *iucvdata* is confused.

System Action: The stage terminates with return code 1279.

User Response: Contact your systems support staff.

System Programmer Response: This is a programming error in *CMS Pipelines*. Investigate whether corrective service is available.

1281E Unsupported IUCV message format

Explanation: A message pending or message complete interrupt is received that either requests a reply or has an incorrect message class.

System Action: The interrupt parameters are displayed. The stage terminates with return code 1281.

1282E Error number on HNDIO

Explanation: *CMS Pipelines* was unable to establish an interrupt handler for a virtual device. Refer help for macro *hndio* for a list of return codes.

System Action: The stage terminates with return code 1282.

1283E More than number CCWs in input record

Explanation: An input record contains sixteen or more control CCWs.

System Action: The stage terminates with return code 1283.

1284E Subchannel for device number hex is busy

Explanation: The device is not available to start a channel program. Condition code 2 was set on the start subchannel instruction.

System Action: The stage terminates with return code 1284.

1285I Input stream number is only stream connected

Explanation: Tracing message. SELECT ANYINPUT is issued. Return code 4 is being set as there is only one input stream left connected.

1286E SF4 is not specified

Explanation: EMSGSF4 is specified, but SF4 has not been specified or has been quietly overridden by an other deblocking option.

System Action: The stage terminates with return code 1286.

1287E Server responds without SF4

Explanation: *tcpclient* with EMSGSF4 is specified and the first byte of data from the server is not zero in the leftmost five bits. This can be caused by *inetd* issuing error messages, in conjunction with starting the server.

System Action: The data stream is converted from ASCII to EBCDIC, deblocked, and then issued with message 39 by a separate stage. On end-of-file from the server, the stage terminates with return code 1287.

1288I Branch to zero probably from *hex*

Explanation: A program check has occurred while executing an instruction at location zero in storage. Register 14 indicates a branch and link instruction and register 15 is zero.

The branch to zero may be due to the branch and link, but it is also possible that the zero in register 15 is the return code from a subroutine. In this case, the branch must have been subsequent to the subroutine call, or from the subroutine.

1289E Third level interrupt exit is already set at *hex*

Explanation: A different exit address was specified for an external interrupt that already has an exit established.

System Action: The stage terminates with return code 1289.

1291I The field ADMSCWR in NUCON is incorrect; found *hex*; display of ABEND information may be in jeopardy

Explanation: This message is issued when *CMS Pipelines* initialises itself and finds that the address in NUCON of the console write routine does not point into the CMS nucleus, as defined by the fields *nuca* and *nucsigma*. This condition may occur when *CMS Pipelines* is initialised under control of other programs that trap console output.

CMS Pipelines will not be able to issue meaningful diagnostic messages in the event of a CMS ABEND while *cms* or *command* is running.

User Response: Initialise the *CMS Pipelines* explicitly in the virtual machine's profile. This may be accomplished by PIPE HOLE.

1296I ABEND in CMS command. Last *number* lines of output follow

Explanation: A command issued through *command* or *cms* has ended abnormally.

System Action: The command output is discarded, except that the last five lines of output are displayed.

1297I Trace table at *hex*

Explanation: Debugging message when an IUCV trace table is present and one of the tracing flags are on. The format of the contents of the trace table is unspecified.

1298E Binary number too large for counter (reason *number*)

Explanation: The input field contains too many significant digits to fit within the 31 decimal digits available in a counter.

System Action: The stage terminates with return code 1298.

1299W *number* duplicate masters were discarded

Explanation: The input master file contained duplicates. This is unlikely to be what you desire.

1300E Time zone offset *number* is not valid (86399 is max)

Explanation: A time zone offset is numerically larger than the number of seconds in a day.

System Action: The stage terminates with return code 1300.

1301E Not a built-in function: *word*

Explanation: A string of characters and digits is met, but it does not name any of the built-in functions.

System Action: The stage terminates with return code 1301.

1302E Leftmost word of 32-bit counter *number* is not zero (*hex*)

System Action: The stage terminates with return code 1302.

1303E Function name expected, but identifier found: *number*

Explanation: The 407 emulator does not support identifiers as variables. Specify either a single character for a field identifier or the name of a built-in function with a suffixed left parenthesis.

System Action: The stage terminates with return code 1303.

1306E • 1320E

1306E First record on track not 5 bytes long (it is *number*)

Explanation: The deblocked track must begin with a pseudo home address, which is five bytes long.

System Action: The stage terminates with return code 1306.

1307E Track capacity exceeded

Explanation: The length of the track being built is larger than 64K, which is the architectural maximum for a CKD track.

System Action: The stage terminates with return code 1307.

1308I Device *hex* is busy or has interrupt pending

Explanation: Return code 5 is received on a Diagnose A8 instruction.

System Action: The operation is retried up to four times. If the operation cannot be started, the stage terminates with return code 1308.

1309E Undefined return code *number* from Diagnose A8 on device *hex*

Explanation: An undocumented return code is received on a Diagnose A8 instruction.

System Action: The stage terminates with return code 1309.

User Response: Contact your systems support staff.

System Programmer Response: This would appear to be a change in the Control Program. Investigate the meaning of the return code in the documentation of Diagnose A8 in the latest edition of *CP Programming Services*.

1310I Device *hex* has unsolicited status pending

Explanation: Return code 16 is received on a Diagnose A8 instruction.

System Action: The operation is retried up to four times. If the operation cannot be started, the stage terminates with return code 1310.

1311I Not squished track reason *hex*

Explanation: A debug message.

1312E Filter package *word* is already loaded

System Action: The stage terminates with return code 1312.

1313E PTF filter package *word* is already loaded

System Action: The stage terminates with return code 1313.

1314E Unable to load module *word* (return code *number*)

System Action: The stage terminates with return code 1314.

1315E Filter package *word* has bad eye-catcher *word*

User Response: The filter package must be linked with the object module FPLNXG.

System Action: The stage terminates with return code 1315.

1316E Filter package *word* is not loaded

Explanation: The specified filter package is not known to *CMS Pipelines*. On CMS, it is neither loaded actively nor passively.

System Action: The stage terminates with return code 1316.

1317E Filter package *word* is not loaded by FILTERPACK LOAD

Explanation: A filter package with the specified name exists, but it was not installed by the *fltpack* stage.

User Response: Use the NUCXDROP command to drop the filter package.

System Action: The stage terminates with return code 1317.

1318E Filter package *word* is in use by *number* stages

System Action: The stage terminates with return code 1318.

1319E Filter package cannot be loaded globally (task is not job step)

Explanation: *fltpack* LOAD GLOBAL was specified, but the task is not the job step task.

System Action: The stage terminates with return code 1319.

1320E Module *word* contains a type 1 filter package; run it as a CMS command to install

Explanation: The filter package cannot be loaded explicitly.

User Response: Invoke the module as a CMS command to load the filter package. On z/OS, link the module with FPLNXG rather than with FPLNXF.

System Action: The stage terminates with return code 1320.

1321I Assembler requests *number* bytes output for record *number* on stream *number*

Explanation: The High Level Assembler has called an exit to write a record, but the exit request information contains a negative length, which is displayed in the message.

System Action: The request is ignored.

1322I Ignoring HALT at *hex*

Explanation: A specification exception is recognised and the instruction is a diagnose with code 8 and a length of 0. This is used on CMS to put the virtual machine into console function mode so the user can inspect storage and registers and in general use CP debugging facilities.

System Action: Retry is attempted. If the system allows, execution continues with the next sequential instruction after the diagnose. In effect, the halt is ignored.

1323E Expression evaluated to the string "*string*"

Explanation: The expression was parsed as returning a number, but the actual result is a string.

System Action: Processing terminates with return code 1323.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *CMS Pipelines*.

1324E Expression evaluated to the number "*hex*"

Explanation: The expression was parsed as returning a string, but the actual result is a number, which is substituted as a hexadecimal dump of its internal representation.

System Action: Processing terminates with return code 1324.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *CMS Pipelines*.

1325E Unrecognised option *string*

Explanation: The second argument to the `strip()` function does not begin with "b", "l", or "t" (in either upper or lower case).

System Action: Processing terminates with return code 1325.

1326E Pad is not a single character (it is *string*)

Explanation: The third argument to the `strip()` function is a string with more than one byte.

System Action: Processing terminates with return code 1326.

1327E Scanner jammed in state *number* in start condition *number*

Explanation: The expression does not contain a valid sequence of characters. The substituted numbers are useful only when debugging *CMS Pipelines*.

System Action: Processing terminates with return code 1327.

1328E There is no default for the type argument

Explanation: The second argument to `DATATYPE` is a null string. You must specify at least one character.

System Action: Processing terminates with return code 1328.

1329E Attempt to extract the square root of a negative number

Explanation: Note that arithmetic in *specs* is carried out using approximate numbers. As a result, the normal laws of algebra do not all hold. For example, $1 - (1/3) * 3$ will not be zero (it will be negative), whereas $1 - (1 * 3) / 3$ will be zero.

System Action: Processing terminates with return code 1329.

1330E Return code *number* on diagnose E0 subcode *hex*

System Action: Processing terminates with return code 1330.

1331E SPOOL file *number* does not exist

Explanation: The specified SPOOL file is not available to the virtual machine or it is not a trace file.

User Response: Use CP command "query trf *" to display the trace files available to you.

System Action: Processing terminates with return code 1331.

1332E SPOOL file *number* contains CP trace data

Explanation: The specified SPOOL file is not the requested format.

User Response: Use CP command "query trf *" to display the trace files available to you and their file types.

System Action: Processing terminates with return code 1332.

1333E • 1352E

1333E SPOOL file number does not contain CP trace data

Explanation: The specified SPOOL file is not the requested format.

User Response: Use CP command “query trf *” to display the trace files available to you and their file types.

System Action: Processing terminates with return code 1333.

1334E SPOOL file number is in use

Explanation: The specified SPOOL file is open by another user or in another stage.

User Response: If a pipeline ends abnormally while reading trace data, the file will not be closed and this message is issued on a subsequent attempt to read the trace file. It may be necessary to reset the virtual machine; that is, IPL CMS.

System Action: Processing terminates with return code 1334.

1335W Concatenated data set(s) for DD=DDNAME ignored. Use QSAM instead

Explanation: Two or more concatenated input data sets are specified on the allocation and a member is requested from the first one. Subsequent data sets in the concatenation are ignored as errors in their specification cannot be verified by CMS Pipelines.

User Response: Replace the device driver with *qsam*. *qsam* will allow the concatenation (it does not inspect the allocation at all); whether it will process the concatenation is another matter. For example, concatenating a member of a partitioned data set with an entire partitioned data set (which reads its directory) will lead to error messages depending on the record format of the first data set in the concatenation.

System Action: Processing continues with the first data set in the concatenation.

1336E Reason number on string: string

Explanation: A call to WebSphere MQ Series failed with the reason code shown on a call to the function name in the second substitution.

System Action: Processing terminates with return code 1336.

1337E Expect CSQN205I; received string

Explanation: The first response message from the command processor is not the expected one.

User Response: Inspect the substituted message text to determine whether it indicates some other kind of error.

System Action: Processing terminates with return code 1337.

1338E ABEND hex reason number on LOAD of entry point

System Action: Processing terminates with return code 1338.

1339E Error opening string for string

Explanation: A queue could not be opened.

User Response: Look for accompanying RACF messages that indicate missing authorisation.

System Action: Processing terminates with return code 1339.

1340E message

Explanation: The MQ command processor has indicated an error in processing the command.

User Response: Refer to the documentation of the message substituted (CSQN205I). If the return code is 20, you are not authorised to issue commands through the system command queue.

System Action: Processing terminates with return code 1340.

1341I data data

1342I data data data

1343I data data data data

1344I data data data data data

1345I data data data data data data

1346I data data data data data data data

1347I data data data data data data data data

1348I data data data data data data data data data

1349I data data data data data data data data data data

Explanation: Messages used for tracing and debugging.

1350E Already connected to queue manager string

Explanation: Another stage is active with the specified queue manager.

System Action: Processing terminates with return code 1350.

1351E Key length number is not valid

Explanation: The record containing the key must be eight, sixteen, or twenty-four bytes for *cipher* DES; it must be sixteen or twenty-four bytes for *cipher* 3DES. For *cipher* AES the key must be sixteen, twenty-four or thirty-two bytes.

System Action: Processing terminates with return code 1351.

1352E Cipher Message instruction not available

System Action: Processing terminates with return code 1352.

1353E Cipher functions are not available *hex*

Explanation: Hardware support for the combination of cipher function and key length is not available.

System Action: Processing terminates with return code 1353.

1354E Computed output column is not positive (it is number)

System Action: Processing terminates with return code 1354.

1355E Unable to convert to integer. number digits in fraction

Explanation: The result of an expression is being converted to integer for use in specifying a position.

System Action: Processing terminates with return code 1355.

1356E Unable to convert to integer: number

Explanation: The result of an expression is being converted to integer for use in specifying a position. The number has more significant digits than can be represented in a 32-bit integer.

System Action: Processing terminates with return code 1356.

1357E Unable to convert to integer. Exponent too large. (number)

Explanation: The result of an expression is being converted to integer for use in specifying a position. The exponent is too large to convert to a 31-digit number.

System Action: Processing terminates with return code 1357.

1358W Global lock held by R12=address R14=address

Explanation: The message level for thorough dispatcher checks is on and the dispatcher is called by a stage that holds the global lock.

The message is issued only the first time that the condition is detected in all concurrently active pipeline sets, as this supposedly is the point of failure. Further dispatching activity is likely to lead to continued detection until the original holder of the lock releases it.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *CMS Pipelines* if the stage is a built-in program.

For stages written by a user: Congratulations on managing to obtain the global lock. Holding the lock when calling the

dispatcher is not a good idea as the dispatcher may switch to another stage. If that stage adds a pipeline specification to the pipeline set, the stage resolution process also acquires the global lock and an assert error 128 results.

System Action: Processing continues. Message 411 is issued if the procedure can be identified. In particular, the lock is not released as that would lead to an assert error 129 when the lock is released by the code that obtained the lock.

1359E Record length number not multiple of cipher block size number

System Action: Processing terminates with return code 1359.

1360E Degenerate Triple DES key

Explanation: The key is 16 or 24 bytes, consisting of two or three keys of eight bytes each. Both or all keys should not be equal as this would degenerate the algorithm to single DES.

User Response: Specify CIPHER DES if you wish to downgrade to single DES. Performance may improve by specifying eight bytes key.

System Action: Processing terminates with return code 1360.

1361E IEWBFDAT code code returns code number reason X'hex'

Explanation: An error is reported by the binder fast data interface.

User Response: Refer to the reason codes in chapter 6 of *MVS Program Management: Advanced Facilities*, SA22-7644.

System Action: Processing terminates with return code 1361.

1362E Unable to load module word (ABEND code HEX reason number cause number)

Explanation: The cause describes the error condition:

0 OK; LOAD macro completed OK. It is a programming error in *CMS Pipelines* if this reason code is displayed.

1 The contents of general register 0 is not correct. This is a programming error in *CMS Pipelines*.

2 General register 2 does not contain an assigned number. This is a programming error in *CMS Pipelines*. (FPLOSM likely needs reassembly.)

3 LOAD instruction failed. The return code describes the error.

System Action: The stage terminates with return code 1362.

1363E • 1377E

: 1363E Odd string length *number*

: **System Action:** The stage terminates with return code
: 1363.

: 1364W Member *word* has no sections

: **Explanation:** Reason code X'10800062' is returned by
: IEWBFDAT.

: **System Action:** Processing continues. *bfda* writes a null
: record to both its streams and continues with the next
: member.

: 1365E Warp *word* not registered

: **Explanation:** The stage is not first in a pipeline, but the
: specified warp ID has not been registered by a *warp* stage
: that is first in a pipeline.

: **System Action:** The stage terminates with return code
: 1365.

: 1366E Warp *word* already registered

: **Explanation:** The stage is first in a pipeline, but the
: specified warp ID has already been registered by another
: *warp* stage that is first in a pipeline.

: **System Action:** The stage terminates with return code
: 1366.

: 1367E Warp *word* no longer registered

: **Explanation:** The warp has terminated.

: **System Action:** The stage terminates with return code
: 1367.

: 1368E Format character '*char*' not valid

: **System Action:** The stage terminates with return code
: 1368.

: 1369E IDR record does not begin X'80'; found X'*char*'

: **System Action:** The stage terminates with return code
: 1369.

: 1370E IDR record indicates *number* bytes present, but : record is *number*

: **Explanation:** The record contains X'80' in column 1, but
: column 2 does not contain one less than the record length.

: **System Action:** The stage terminates with return code
: 1370.

: 1371E Improper IDR language processor flag byte X'*hex*' : at offset *number*

: **Explanation:** The byte should be 0 or 1, but it is not. The
: offset is relative to the first CESD number for a particular
: control section; not to the beginning of a particular record, as
: this type of identification data is spanned across records.

: **System Action:** The stage terminates with return code
: 1371.

: 1372E Improper control record prefix *hex*

: **Explanation:** One of the two length fields in the control
: record contains a number that is not a multiple of four.

: **System Action:** The stage terminates with return code
: 1372.

: 1373E Control record requests *number* bytes, but only : *number* bytes are available

: **System Action:** The stage terminates with return code
: 1373.

: 1374E CESD IDs descending *number* follows *number*

: **System Action:** The stage terminates with return code
: 1374.

: 1375E CTL or RLD record expected, but found *hex*

: **Explanation:** A control record has indicated that a number
: of relocation dictionary records will follow the text record,
: but some other kind of record was found. The load module
: is probably broken.

: **System Action:** The stage terminates with return code
: 1375.

: 1376E RLD record expected, but found CTL *hex*

: **Explanation:** A control record has indicated that a number
: of relocation dictionary records will follow the text record,
: but some other kind of record was found. The load module
: is probably broken.

: **System Action:** The stage terminates with return code
: 1376.

: 1377E CTL record found as record X'*hex*', but count is : X'*hex*'

: **Explanation:** A control record has indicated that a number
: of relocation dictionary records will follow the text record,
: but some other kind of record was found after that run. The
: load module is probably broken.

: This is rather vague because the load module format
: provides a one byte count, but more than 256 RLD has been
: observed in the wild. However, the modulo 256 does not
: agree.

: **System Action:** The stage terminates with return code
: 1377.

: **1378E Installation validation routine rejected SVC 99**

: **Explanation:** An installation exit has denied dynamic allo-
: cation.

: **System Action:** The stage terminates with return code
: 1378.

: **User Response:** Contact your systems support staff.

: **1379E Unexpected return code X'hex' on SVC 99**

: **Explanation:** The return code was not one of the docu-
: mented ones.

: **System Action:** The stage terminates with return code
: 1379.

: **User Response:** Contact your systems support staff.

: **1380E Data set *string* is not a program library; member
: *word***

: **Explanation:** IEWBFDAT returns code 4 reason
: X'10800029', which indicates that the data set is not a
: proper program library or the selected member is broken.

: **System Action:** The stage terminates with return code
: 1380.

: **User Response:** Contact your systems support staff.

: **1381I Pipeline *word* committed to *number* worst return
: *code number***

: **Explanation:** Pipeline dispatcher trace is active. The pipe-
: line specification commits to the level shown.

: **System Action:** None.

: **1382E Tertiary stream not defined**

: **Explanation:** The primary stream and the secondary stream
: are defined.

: **System Action:** The stage terminates with return code
: 1382.

: **1383E Unexpected EOF on primary input**

: **Explanation:** The primary stream comes to end-of-file
: without having presented a control record indicating end of
: module.

: **System Action:** The stage terminates with return code
: 1383.

: **1384E Structure name expected; found *string***

: **Explanation:** A structure specifier is expected at the begin-
: ning of the input.

: **System Action:** The stage terminates with return code
: 1384.

: **1385E Structure *name* is empty**

: **Explanation:** A colon is the next nonblank character after
: the structure name or there is no further input.

: **System Action:** The stage terminates with return code
: 1385.

: **1386E No structure name found**

: **Explanation:** End-of-file was met after a colon indicating
: the beginning of the definition of a structure.

: **System Action:** The stage terminates with return code
: 1386.

: **1387E Incorrect first character in identifier: *string***

: **Explanation:** Structure and member names (often referred
: to as identifiers) must begin with a letter in the English
: alphabet or one of the special characters “@#\$!?” (at sign,
: number sign, dollar sign, exclamation point, question mark,
: and underscore). The second and subsequent character may
: also be a digit.

: Identifiers are case sensitive unless the structure is defined as
: caseless.

: **System Action:** The stage terminates with return code
: 1387.

: **1388E Structure already defined: *name***

: **System Action:** The stage terminates with return code
: 1388.

: **1389E Member already defined: *name***

: **System Action:** The stage terminates with return code
: 1389.

: **1390E Incomplete member definition: *name***

: **Explanation:** A member name or hyphen is found, but the
: definition is missing. Either end-of-file or a colon is met.

: **System Action:** The stage terminates with return code
: 1390.

: **1391E "*char*" is not valid in identifier *string***

: **System Action:** The stage terminates with return code
: 1391.

1392E • 1407E

: **1392E Structure not defined:** *name*
: **System Action:** The stage terminates with return code
: 1392.

: **1393E No structures defined in pipeline set**
: **System Action:** The stage terminates with return code
: 1393.

: **1394E Structure still in use:** *name (number users)*
: **Explanation:** An attempt is made to delete a structure that
: is embedded in another structure.

: **User Response:** Delete the embedding structure first or
: pass the two structure names on the same input line in any
: order.

: **System Action:** The stage terminates with return code
: 1394.

: **1395E Unqualified member name:** *name*
: **Explanation:** No qualifier is active and the member name
: contains no period.

: **System Action:** The stage terminates with return code
: 1395.

: **1396E Incomplete inputRange** *string*
: **Explanation:** A keyword is met that is valid in an
: *inputRange*, but no columns or members are specified.

: **System Action:** The stage terminates with return code
: 1396.

: **1397E Missing identifier in qualified name:** *word*
: **Explanation:** The last character of the word is a period.

: **System Action:** The stage terminates with return code
: 1397.

: **1398E Member *name* not defined in structure *name***
: **Explanation:**

: **System Action:** The stage terminates with return code
: 1398.

: **1399E Member name further qualified with *name***
: **Explanation:** A scalar member is met in an identifier that
: is continued with a period to indicate a member of a struc-
: ture.

: **System Action:** The stage terminates with return code
: 1399.

: **1400E Structure not further qualified:** *name*
: **Explanation:** No member name is found.

: **System Action:** The stage terminates with return code
: 1400.

: **1401E Qualifier contains member:** *name*
: **Explanation:** A qualifier is requested, but one of the levels
: refer to a member that is not an embedded structure.

: **System Action:** The stage terminates with return code
: 1401.

: **1402E No structures defined in thread**
: **System Action:** The stage terminates with return code
: 1402.

: **1403E Premature end of expression; term expected**
: **Explanation:** An operator or left parenthesis is met at the
: end of the expression.

: **System Action:** The stage terminates with return code
: 1403.

: **1404E Floating point number too short (length *number*)**
: **Explanation:** The field length is less than 2.

: **System Action:** The stage terminates with return code
: 1404.

: **1405E Floating point number too long (length *number*)**
: **Explanation:** The field length is greater than 8.

: **System Action:** The stage terminates with return code
: 1405.

: **1406E Fixed number needs at least *number* columns**
: **Explanation:** The output field length is too short to contain
: the number without truncating significant leftmost bits.

: **System Action:** The stage terminates with return code
: 1406.

: **1407E Exponent overflow (*number*)**
: **Explanation:** The input fixed point number is too large to
: convert to a hexadecimal floating point number.

: **System Action:** The stage terminates with return code
: 1407.

-
- : **1408E Length of output member (*number*) is above**
 : **maximum *number***
- : **Explanation:** For fixed point, the limit is 128 bytes; for
 : floating point, it is eight.
- : **System Action:** The stage terminates with return code
 : 1408.
-
- : **1409E Counter exponent out of range for hexadecimal:**
 : ***number***
- : **System Action:** The stage terminates with return code
 : 1409.
-
- : **1410E No record read from stream *number***
- : **Explanation:** End-of-file was received when trying to read
 : a record.
- : **System Action:** The stage terminates with return code
 : 1410.
-
- : **1411E Too few streams are defined; *number* are present,**
 : **but three streams are needed**
- : **System Action:** The stage terminates with return code
 : 1411.
-
- : **1412E Allocation would require more than two gigabytes**
- : **System Action:** The stage terminates with return code
 : 1412.
-
- : **1413E Found *number* columns**
- : **System Action:** The stage terminates with return code
 : 1413.
-
- : **1414E Found *number* rows**
- : **System Action:** The stage terminates with return code
 : 1414.
-
- : **1415E Record length *number* is not a multiple of four**
 : **(stream *number*)**
- : **System Action:** The stage terminates with return code
 : 1415.
-
- : **1416E Null record read from stream *number***
- : **System Action:** The stage terminates with return code
 : 1416.
-
- : **1417E String length cannot be negative: *number***
- : **System Action:** The stage terminates with return code
 : 1417.
-
- : **1418E String position cannot be zero**
- : **System Action:** The stage terminates with return code
 : 1418.
-
- : **1419E Too few arguments in function call**
- : **System Action:** The stage terminates with return code
 : 1419.
-
- : **1420E Too many arguments in function call**
- : **System Action:** The stage terminates with return code
 : 1420.
-
- : **1421E DO expected; *word* was found**
- : **Explanation:** A condition expression has been scanned
 : after WHEN, but there is no further data or the next word is
 : not DO.
- : **System Action:** The stage terminates with return code
 : 1421.
-
- : **1422E DONE expected; *word* was found**
- : **Explanation:** A condition expression has been scanned
 : after WHILE and DO, but the next word terminates an IF
 : group.
- : **System Action:** The stage terminates with return code
 : 1422.
-
- : **1423E Incomplete WHILE**
- : **System Action:** A WHILE group has been opened, but end-
 : of-file is met without a matching DONE. The stage termi-
 : nates with return code 1423.
-
- : **1424E Counter underflow**
- : **Explanation:** The exponent of a counter has underflowed.
- : **System Action:** The stage terminates with return code
 : 1424.
-
- : **1425W Use parentheses when using the result of an**
 : **assignment: *string***
- : **Explanation:** An operator sees a counter assignment as its
 : right hand operand.
- : **User Response:** Enclose the assignment in parentheses.
- : **System Action:** So far, this is a nuisance message to prod
 : you to fix the expression.

1426I ... Evaluating "string"

Explanation: This message is issued after an error message has been issued by the *spec* expression evaluator and the message level is odd.

System Action: None.

1427E Exponent out of range: number

Explanation: The fixed point binary number is too large for conversion to the internal counter format.

The number can be represented in the internal representation, but the conversion algorithm uses a limited exponent range corresponding to the one that is valid for hexadecimal floating point numbers.

System Action: The stage terminates with return code 1427.

1428E Member name has no type

System Action: The stage terminates with return code 1428.

1429E Member name has unsupported type char

System Action: The stage terminates with return code 1429.

1430E Not hexadecimal: X'string'

System Action: The stage terminates with return code 1430.

1431E Member name longer than 16M (it is number)

System Action: The stage terminates with return code 1431.

1432E Bad placement option string

Explanation: The third parameter in the output placement expression is not CENTRE (CENTER), LEFT, or RIGHT; or an abbreviation of these words.

System Action: The stage terminates with return code 1432.

1433E Computed output length is negative (it is number)

Explanation: A length of zero means take the default length as by the data to be loaded.

System Action: Processing terminates with return code 1433.

1434E Parse error in state number, unexpected string at offset number: "string"

Explanation: The expression does not parse according to the grammar. The state number is of interest only to the author of *CMS Pipelines*; the first string shows the mnemonic name of the input token that the grammar cannot parse.

System Action: Message 1435 is issued. Processing terminates with return code 1434.

1435I Expecting string

Explanation: List the acceptable token names in the current parser state.

System Action: Processing terminates with return code 1435.

1436E FIXED specified, but no record length specified and no input

Explanation: Specify an explicit record length if you really want to create a null file with a particular record length.

System Action: Processing terminates with return code 1436.

1437E Previous member did not establish a position for word

Explanation: The previous member was specified as a word, field, auto field, or length * Its position is not known at the time the structure is defined.

System Action: Processing terminates with return code 1437.

1438I Incorrect text unit type X'hex'

System Action: Further informational messages are issued. Processing terminates eventually.

1439E Left hand operand is a string

Explanation: The left hand operand of an arithmetic operator is a counter that contains a string.

System Action: Processing terminates with return code 1439.

1440E Right hand operand is a string

Explanation: The right hand operand of an arithmetic operator is a counter that contains a string.

System Action: Processing terminates with return code 1440.

-
- : **1441I ... Processed *number* structures and *number***
 : **members in next structure**
- : **Explanation:** Informational message issued when *structure*
 : ADD is terminating because of an error. The first number
 : represent the number of completely finished structure
 : definitions; zero mean that the error is in the first structure or
 : member.
-
- : **1442E Both ranges specify same length as other *string***
- : **Explanation:** The two ranges in the comparison both
 : specify plus as the length; this makes the length indetermi-
 : nate.
- : **System Action:** Processing terminates with return code
 : 1442.
-
- : **1443E Comma list is available only with equal compares**
- : **Explanation:** The right hand range specifies plus as its
 : length.
- : **System Action:** Processing terminates with return code
 : 1443.
-
- : **1444E Comma list is not available with implied length**
- : **Explanation:** The operator must be = or ==.
- : **System Action:** Processing terminates with return code
 : 1444.
-
- : **1445I Error in call to *function: string***
- : **Explanation:** A built-in function has detected an error.
-
- : **1446E String contains leading or trailing blank: "*string*"**
- : **Explanation:** The argument to X2C must not contain
 : leading or trailing blanks.
- : **System Action:** Processing terminates with return code
 : 1446.
-
- : **1447E String contains blank not on byte boundary:**
 : "*string*"
- : **System Action:** Processing terminates with return code
 : 1447.
-
- : **1448E String contains a character that is not hexadecimal**
 : *char: string*
- : **System Action:** Processing terminates with return code
 : 1448.
-
- : **1449E Pad character is a string, not a single character:**
 : *string*
- : **System Action:** Processing terminates with return code
 : 1449.
-
- : **1450E Option string is null**
- : **Explanation:** If specified, the string must contain at least
 : one character (it can be any positive length).
- : **System Action:** Processing terminates with return code
 : 1450.
-
- : **1451E Option string is not valid for function: *character***
- : **Explanation:** The first character of the option string does
 : not contain a character that is valid for the function.
- : **System Action:** Processing terminates with return code
 : 1451.
-
- : **1452E Argument is a string, not a single character:**
 : *string*
- : **Explanation:** The arguments to the XRANGE function must
 : both be a single character unless they are omitted.
- : **System Action:** Processing terminates with return code
 : 1452.
-
- : **1453I Trap issued the CP command "*string*"**
- : **Explanation:** Informational message to indicate that an
 : activated trap has sprung and a CP command was issued.
 : Most likely there will be a dump of the virtual machine in a
 : reader somewhere.
-
- : **1454E Not valid packed data *hex***
- : **Explanation:** A field that should contain packed decimal
 : data contains an incorrect bit combination.
- : **System Action:** Processing terminates with return code
 : 1454.
-
- : **1455E Output field is *number* bytes, but packed number**
 : **requires *number* bytes to avoid truncation**
- : **Explanation:** A counter is converted to packed decimal
 : integer.
- : **System Action:** Processing terminates with return code
 : 1455.
-
- : **1456E Scale not numeric: *string***
- : **Explanation:** A member type character is met followed by
 : a left parenthesis, but the word in the parentheses is not a
 : number
- : **System Action:** Processing terminates with return code
 : 1456.

1457E • 1470E

: **1457E Scale out of bounds:** *number (-32768 to 32767 is valid range)*

: **Explanation:** A member type character is met followed by a left parenthesis, the word in the parentheses is a number, but it is too large.

: **System Action:** Processing terminates with return code 1457.

: **1458E Semicolon expected; found** *string*

: **Explanation:** A prefix or suffix specification does not end at a semicolon.

: **System Action:** Processing terminates with return code 1458.

: **1459E Plus or minus expected; found** *string*

: **Explanation:** A prefix or suffix specification must indicate whether matching it means to write a hit record (plus) or not (minus).

: **System Action:** Processing terminates with return code 1459.

: **1460E Incomplete pattern**

: **Explanation:** A prefix or suffix specification is not ended in a comma or semicolon; an optional pattern list does not end in >.

: **System Action:** Processing terminates with return code 1460.

: **1461E Missing pattern at** *string*

: **System Action:** Processing terminates with return code 1461.

: **1462E Comma expected; found** *string*

: **Explanation:** A prefix or suffix specification does not end at a comma or semicolon.

: **System Action:** Processing terminates with return code 1462.

: **1463E No matching specified**

: **Explanation:** A hyphen is specified for both the prefix and the postfix pattern (an omitted postfix is treated like a hyphen).

: **System Action:** Processing terminates with return code 1463.

: **1464E Odd number of nibbles** (*number*) **in pattern:** *string*

: **Explanation:** You guessed it. The pattern must cover complete bytes.

: **User Response:** If you desire an odd number of matched nibbles, add a “don’t care” nibble (a period).

: **System Action:** Processing terminates with return code 1464.

: **1465E Missing number at end of pattern:** *string*

: **Explanation:** An ampersand ends the pattern. It must have a number to specify the register to hold the nibble.

: **System Action:** Processing terminates with return code 1465.

: **1466E Pattern longer than 32767 bytes**

: **Explanation:** The pattern to match or the control structure to describe it is too long.

: **System Action:** Processing terminates with return code 1466.

: **1467E Expect >; found** *char*

: **Explanation:** An optional item list was opened, but it was terminated by a comma or semicolon.

: **System Action:** Processing terminates with return code 1467.

: **1468E Semicolon, colon, or comma expected; found** *char*

: **Explanation:** An required item list ends in >.

: **System Action:** Processing terminates with return code 1468.

: **1469E Unexpected end of module** *word*

: **Explanation:** End-of-file or a null record is read. The module should end with an end of module flag bit in a control record.

: **System Action:** Processing terminates with return code 1469.

: **1470E CCW length** *number* **differs from record length** *number module word*

: **Explanation:** The control record specifies a different length than the actual text record that follows. The module being read is broken.

: **System Action:** Processing terminates with return code 1470.

-
- : **1471E Unrecognised STOP parameter:** *word*
- : **Explanation:** PIPMOD STOP is issued, but the additional keyword is not ACTIVE.
- : **System Action:** Processing terminates.
-
- : **1472E Unrecognised PIPMOD immediate command:**
: *word*
- : **Explanation:** PIPMOD is issued, but the subcommand is not valid.
- : **System Action:** Processing terminates.
-
- : **1473E Unable to obtain global lock; held by** *hex*
- : **Explanation:** The global lock is held, which makes further processing impossible. The address of the lock control word is substituted in the message.
- : **System Action:** Processing terminates.
-
- : **1474I Global** *hex*
- : **1475I Thread** *hex*
- : **1476I Header** *hex*
- : **1477I Vector** *hex*
- : **1478I Stage** *hex*
- : **Explanation:** Messages issued in response to the immediate command PIPMOD WHERE that display the pipeline control blocks
-
- : **1479I Running:** *string*
- : **Explanation:** Message issued in response to the immediate command PIPMOD ACTIVE.
-
- : **1480I In procedure** *word*
- : **Explanation:** Message issued in response to the immediate command PIPMOD ACTIVE.
-
- : **1481I Stage is flagged to stop. PSW not in CMS Pipelines code**
- : **Explanation:** PIPMOD STOP ACTIVE was issued. The active stage has been flagged to stop next time it enters the dispatcher. The instruction address of the I/O PSW does not point to *CMS Pipelines* code.
-
- : **1482I Stage is in the dispatcher; likely to stop on the way out.**
- : **Explanation:** PIPMOD STOP ACTIVE was issued. The active stage has been flagged to stop. The instruction address of the I/O PSW points into the dispatcher; the stage is likely to terminate immediately.
-
- : **1483I Stage is flagged to stop. PSW in wait/free storage management.**
- : **Explanation:** PIPMOD STOP ACTIVE was issued. The active stage has been flagged to stop next time it enters the dispatcher. The instruction address of the I/O PSW points to the wait routine or the free storage manager.
-
- : **1484I Stage is flagged to stop. It is not summarily stoppable**
- : **Explanation:** PIPMOD STOP ACTIVE was issued. The active stage has been flagged to stop next time it enters the dispatcher. The stage cannot be stopped at this point, but it is likely to terminate next time it calls the dispatcher.
-
- : **1485I Stage is flagged to stop. I/O old PSW and IOPSW fields are not the same. Type B to continue**
- : **Explanation:** PIPMOD STOP ACTIVE was issued. The active stage has been flagged to stop next time it enters the dispatcher. The stage cannot be stopped at this point because the CMS I/O information is in an inconsistent state. A CP read has been put up to allow you to examine the CMS control blocks and low core.
-
- : **1486I Stage is flagged to stop. Forcing exit from** *word*
- : **Explanation:** PIPMOD STOP ACTIVE was issued. The PSW has been modified to force the stage to return.
-
- : **1487E Checksum field in column** *number* **is not within record length** *number*
- : **Explanation:** The field to receive the checksum is not present in the input record.
- : **System Action:** Processing terminates with return code 1487.
-
- : **1488E Convert index** *number* **is not implemented**
- : **Explanation:** The conversion routine selected is not present in *CMS Pipelines*.
- : **User Response:** Contact your systems support staff.
- : **System Programmer Response:** This is an error in *CMS Pipelines*.
- : **System Action:** Processing terminates with return code 1488.
-
- : **1489E Unable to convert from negative to unsigned**
- : **Explanation:** Conversion to unsigned binary is requested, but the input number is negative.

1490I • 1503E

: **1490I Processing item number *number*: *string***
: **Explanation:** Informational message from *spec* when it
: terminates due to a a run time error. Specification items are
: numbered from zero. The first item is a SELECT item gener-
: ated internally to select the primary input stream.

: **1491E Field identifier specified, but no further operands
: are present**

: **System Action:** Processing terminates with return code
: 1491.

: **1492E Field identifier specified, but no valid range found:
: *word***

: **System Action:** Processing terminates with return code
: 1492.

: **1493E Too few streams are defined; *number* are present,
: but *number* are required**

: **System Action:** The stage terminates with return code
: 1493.

: **1494E Equal sign expected; end of member found**

: **Explanation:** A left parenthesis is met indicating that a list
: of manifest constants is to follow and an identifier has been
: scanned, but end of input or a colon was met where an equal
: sign is expected.

: **System Action:** Processing terminates with return code
: 1494.

: **1495E Equal sign expected; found *char***

: **Explanation:** A left parenthesis is met indicating that a list
: of manifest constants is to follow and an identifier has been
: scanned, but the next non-blank character is not an equal
: sign.

: **System Action:** Processing terminates with return code
: 1495.

: **1496E Number expected; end of member found**

: **Explanation:** A left parenthesis is met indicating that a list
: of manifest constants is to follow and an identifier has been
: scanned as well as an equal sign, but end of input or a colon
: was met where a number is expected.

: **System Action:** Processing terminates with return code
: 1496.

: **1497E Comma or right parenthesis expected; end of
: member found**

: **Explanation:** A complete manifest constant has been
: scanned, but end of input or a colon was met where a
: comma or a right parenthesis is expected.

: **System Action:** Processing terminates with return code
: 1497.

: **1498E Comma or right parenthesis expected; found *char***

: **Explanation:** A complete manifest constant has been
: scanned, but end of input or a colon was met where a
: comma or a right parenthesis is expected.

: **System Action:** Processing terminates with return code
: 1498.

: **1499E Member *word* is a manifest constant**

: **Explanation:** A complete manifest constant has been
: scanned, but it is not valid in the context. This includes
: specifying a manifest constant:

- After MEMBER in a structure definition.
- After a field identifier in *spec*.
- After SUBSTR.
- With string compare in *pick*.

: **System Action:** Processing terminates with return code
: 1499.

: **1500E Member *word* is not a manifest constant (it is a
: data member)**

: **Explanation:** A reference to a member is found in the
: array bounds for a member. It must be number that is zero
: or positive, or a manifest constant.

: **System Action:** Processing terminates with return code
: 1500.

: **1501E Right parenthesis expected after array bound;
: found *char***

: **System Action:** Processing terminates with return code
: 1501.

: **1502E Word-style not supported with an array**

: **System Action:** Processing terminates with return code
: 1502.

: **1503E Array size is greater than 2G**

: **System Action:** Processing terminates with return code
: 1503.

-
- : **1504E Index missing for member** *word*
- : **Explanation:** A member has been resolved to an array and a left parenthesis is scanned indicating that an index is present, but no further data are present.
- : **System Action:** Processing terminates with return code 1504.
-
- : **1505E Right parenthesis expected after index**
- : **System Action:** Processing terminates with return code 1505.
-
- : **1506E Index number is out of bounds** (*number*)
- : **System Action:** Processing terminates with return code 1506.
-
- : **1507E Cannot access entire varying bounds array** *word*
 : An member is requested without specifying an index, but the member is defined as an array without specific bound.
- : **System Action:** Processing terminates with return code 1507.
-
- : **1508E Member** *word* **is an array** An member is requested without specifying an index, but the member is defined as an array.
- : **System Action:** Processing terminates with return code 1508.
-
- : **1509E Member** *word* **is a scalar** A member is requested with an index expression, but the member is not defined as an array.
- : **System Action:** Processing terminates with return code 1509.
-
- : **1510E Index number is not positive**
- : **System Action:** Processing terminates with return code 1510.
-
- : **1511E Incomplete subscript in** *string*
- : **Explanation:** A qualifier or member name is being scanned. A left parenthesis has been met, but no matching right parenthesis is found.
- : **System Action:** Processing terminates with return code 1511.
-
- : **1512E "char" is not valid in subscript of identifier** *string*
- : **Explanation:** The subscript must consist of digits only.
- : **System Action:** The stage terminates with return code 1512.
-
- : **1513E Expect period after subscript of identifier** *string*;
 : found *word*
- : **Explanation:** The subscript must consist of digits only.
- : **System Action:** The stage terminates with return code 1513.
-
- : **1514E Subscript "word" is not valid in identifier** *string*
- : **Explanation:** The member must be subscripted. The subscript must consist of digits only. It must also evaluate to 1 or more and not larger than the array bound.
- : **System Action:** The stage terminates with return code 1514.
-
- : **1515E Top level structure "word" cannot be subscripted**
- : **Explanation:** A structure name cannot be subscripted.
- : **System Action:** The stage terminates with return code 1515.
-
- : **1516E Last character of identifier is a period:** *word*
- : **Explanation:** A member name is required after the period.
- : **System Action:** The stage terminates with return code 1516.
-
- : **1517E No active qualifier for** *word*
- : **Explanation:** A single period is specified at the beginning of a member name to indicate that the current qualifier must be used, but no qualifier has been established for the stream.
- : **System Action:** The stage terminates with return code 1517.
-
- : **1518E No identifier found**
- : **Explanation:** The keyword MEMBER was specified, but the member name consists of one or two periods only.
- : **System Action:** The stage terminates with return code 1518.
-
- : **1519E Address space name longer than 24:** *word*
- : **System Action:** The stage terminates with return code 1519.
-
- : **1520E** Return code *number* on
 : ADRSPACE/ALSERV/MAPMDISK diagnose
- : **System Action:** The stage terminates with return code 1520.

1521E • 1535E

: **1521E ALET *hex* is not valid**

: **Explanation:** The ALET supplied on input will cause a program check, if used (*alserv* TEST) or return code 12 is set on ALSERV REMOVE to indicate that the ALET is malformed.

: **System Action:** The stage terminates with return code 1521.

: **1522E Virtual machine is not in XC mode**

: **Explanation:** Validating the ALET supplied on input will cause a program check for special operation exception.

: **System Action:** The stage terminates with return code 1522.

: **1523E ASIT *hex* is not valid**

: **Explanation:** The ASIT supplied on input does not identify an address space owned by the virtual machine. (Return code 4 on ADRSPACE PERMIT)

: **System Action:** The stage terminates with return code 1523.

: **1524E VCIT *hex* does not represent a user that is logged in**

: **Explanation:** The VCIT specified does not represent the primary space of a user that is currently logged on. (Return code 28 on ADRSPACE PERMIT)

: **System Action:** The stage terminates with return code 1524.

: **1525E User *word* is not logged on**

: **Explanation:** (Return code 28 on ADRSPACE PERMIT)

: **System Action:** The stage terminates with return code 1525.

: **1526E Virtual machine may not share address spaces**

: **Explanation:** (Return code 32 on ADRSPACE PERMIT)

: **System Action:** The stage terminates with return code 1526.

: **1527E Address space *word* is not available for user *word***

: **Explanation:** Either the address space does not exist or you have not been granted access rights. Note that the access rights do not survive an IPL in the virtual machine that granted the permission. (Return code 4 on ADRSPACE QUERY)

: **System Action:** The stage terminates with return code 1527.

: **1528E Address space name *word* is not valid**

: **Explanation:** The name contains a character that is not valid in an address space name. (Return code 16 on ADRSPACE)

: **System Action:** The stage terminates with return code 1528.

: **1529E Address space name *word* already exists**

: **Explanation:** (Return code 4 on ADRSPACE CREATE)

: **System Action:** The stage terminates with return code 1529.

: **1530E Maximum number of address spaces is exceeded**

: **Explanation:** Note that your quota is zero unless the user directory entry for your virtual machine contains a xconfig adrspace statement. (Return code 8 on ADRSPACE CREATE)

: **System Action:** The stage terminates with return code 1530.

: **1531E Maximum size of address spaces is exceeded**

: **Explanation:** (Return code 12 on ADRSPACE CREATE)

: **System Action:** The stage terminates with return code 1531.

: **1532E Address space size is not valid: *number***

: **Explanation:** The number is larger than 524,288, which is the number of pages in a two gigabyte address space. (Return code 20 on ADRSPACE CREATE)

: **System Action:** The stage terminates with return code 1532.

: **1533W ASIT *hex* is already permitted to user *word***

: **Explanation:** The secondary output stream is not defined. (Return code 24 on ADRSPACE PERMIT)

: **System Action:** Processing continues.

: **1534W ASIT *hex* is already permitted to VCIT *hex***

: **Explanation:** The secondary output stream is not defined. (Return code 24 on ADRSPACE PERMIT)

: **System Action:** Processing continues.

: **1535E Host access list is full**

: **Explanation:** (Return code 4 on ALSERV ADD)

: **System Action:** The stage terminates with return code 1535.

-
- : **1536W ALET *hex* is neither valid nor revoked.**
 : **Explanation:** (Return code 4 on ALSERV REMOVE)
 : **System Action:** Processing continues.
-
- : **1537E Device *hex* is not a reserved minidisk**
 : **Explanation:** (Return code 12 on DISKID)
 : **System Action:** The stage terminates with return code
 : 1537.
-
- : **1538E Device *hex* is not attached**
 : **Explanation:** (Return code 100 on DISKID)
 : **System Action:** The stage terminates with return code
 : 1538.
-
- : **1539E Too many ranges to save**
 : **Explanation:** More than 509 ranges are presented in one
 : input record to *mapdisk* SAVE.
 : **System Action:** The stage terminates with return code
 : 1539.
-
- : **1540E Page number too large: *number***
 : **Explanation:** The number is larger than the maximum
 : number of pages in a data space.
 : **System Action:** The stage terminates with return code
 : 1540.
-
- : **1541E Digit "*character*" is not hexadecimal in string
 : *number***
 : **System Action:** The stage terminates with return code
 : 1541.
-
- : **1542E Hexadecimal string too long: *number***
 : **System Action:** The stage terminates with return code
 : 1542.
-
- : **1543E No minidisk pool has been defined**
 : **Explanation:** (Return code 32 on MAPMDISK DEFINE)
 : **System Action:** The stage terminates with return code
 : 1543.
-
- : **1544W Dispatcher called with address space control *hex***
 : **Explanation:** The pipeline dispatcher is called in access
 : register mode in an XC virtual machine. The dispatcher does
 : not manage access registers or address space control
 : **System Action:** The mode is set to primary space mode.
-
- : **1545E Data space ALET *hex* is not initialised properly;
 : eye-catcher is *word***
 : **Explanation:** A stage that loads data into a data space is
 : invoked with a nonzero ALET operand, but the first part of
 : the data space is not the proper format. In particular, the
 : first eight bytes do not contain the string *fplasi1l*.
 : **System Action:** The stage terminates with return code
 : 1545.
-
- : **1546E Data space ALET *hex* is in use; lock is *word***
 : **Explanation:** A stage that loads data into a data space is
 : invoked with a nonzero ALET operand, but the lock word in
 : the data space is not binary zeros. This indicates that some
 : other stage is also using the address space. The contents of
 : the lock show may give a clue to the name of the stage that
 : holds the lock.
 : **System Action:** The stage terminates with return code
 : 1546.
-
- : **1547W Data space ALET *hex* contains unexpected lock
 : *word***
 : **Explanation:** A stage can load data into a data space is
 : invoked with a nonzero ALET operand. When the is termi-
 : nating it is finds that the lock in the data space has been
 : changed to the value shown.
 : **System Action:** The stage terminates normally.
-
- : **1548E Insufficient space in the data space for *number*
 : bytes**
 : **Explanation:** A stage that loads data into a data space is
 : invoked with a nonzero ALET operand, and the data space is
 : full.
 : **System Action:** The stage terminates with return code
 : 1548.
-
- : **1549E ALET and PGMLIST are incompatible**
 : **System Action:** The stage terminates with return code
 : 1549.
-
- : **1550E Data space ALET *hex* contains unexpected lock
 : *word***
 : **Explanation:** A stage can read data from a data space is
 : invoked with a nonzero ALET operand, but it is found that
 : the lock in the data space contains the lock value shown
 : rather than the expected one.
 : **System Action:** The stage terminates with return code
 : 1550.

1551E • 1562E

1551E Data space ALET *hex* is not locked

Explanation: A stage can read data from a data space is invoked with a nonzero ALET operand, but it is found that the lock in the data space was cleared.

System Action: The stage terminates with return code 1551.

1552E Data space is fetch protected in key *hex* (PSW key *hex*)

Explanation: A stage that accesses a data space will not be able to do so because the data space is fetch protected and the PSW key is nonzero and different from the key of the first frame in the data space.

System Action: The stage terminates with return code 1552.

1553E Data space is write protected in key *hex* (PSW key *hex*)

Explanation: A stage that moves data into a data space will not be able to do so because the data space is fetch protected and the PSW key is nonzero and different from the key of the first frame in the data space.

System Action: The stage terminates with return code 1553.

1554E Data space ALET *hex* cannot be written

Explanation: A stage that moves data into a data space will not be able to do so because of the protection system. The data space can be read, but not written.

System Action: The stage terminates with return code 1554.

1555E Data space ALET *hex* is not accessible

Explanation: A stage that accesses a data space will not be able to do so because of the protection system. (Test protect condition codes 2 or 3.)

System Action: The stage terminates with return code 1555.

1556E Input record too short for complete VMCMHDR (*number* bytes available; 40 required)

System Action: The stage terminates with return code 1556.

1557W VMCF message arrived, but no listener is active

Explanation: A VMCF message arrived that is not a final response message. No stage is listening for such messages.

System Action: The stage terminates with return code 1557.

1558E Unsupported VMCF function code *number*

Explanation: The input to *vmclient* contains an unsupported function code, for example *unauthorize*.

System Action: The stage terminates with return code 1558.

1559E Structure *word* is not built in

Explanation: *struct* BUILD finds an embedded structure that is not built in. As the time of use of the structure is unknown this could lead to a dangling reference.

System Action: The stage terminates with return code 1559.

1560I Scanned member: *string*

Explanation: The string is the part of the member definition in error that has been scanned so far.

1561E Counter number *number* is not valid (valid: *number* to *number*)

Explanation: The subscript to the counter array is out of bounds. Be sure to specify COUNTERS when using counter arrays. No counter array is allocated when the second number is larger than the third.

System Action: The stage terminates with return code 1561.

1562E Incorrect UTF-*number* X'*hex*' reason code *number*

Explanation: The input contains a sequence of characters that are not valid for the UTF format specified. The reason code indicates the error:

- 4 The UTF-8 input is binary zeros, but MODIFIED is specified (U+0000 should be encoded as X'C080').
- 8 The first byte of an UTF-8 encoded character is of the form B'10xxxxxx', which is reserved for additional bytes in a multibyte sequence. Most likely, a multibyte sequence is too long.
- 12 More bytes are required for the character, but the input record is exhausted.
- 16 A byte other than the first for a UTF-8 encoded character has the leftmost bit zero (B'0xxxxxxx'). That is, a multibyte sequence ended prematurely.
- 20 A byte other than the first for a UTF-8 encoded character has the second bit one (B'x1xxxxxx'). That is, a multibyte sequence ended prematurely.
- 24 The first byte of an UTF-8 encoded character contains five leftmost one bits (B'11111xxx'). Such sequences were defined in RFC 2279 for encodings needing more than 21 bits, but retracted in RFC 2279.
- 28 Overlong UTF-8 encoding. Two bytes for a seven bit code point, except for zero when MODIFIED is specified. Also a any longer sequence that could be expressed in fewer bytes.

-
- : 32 A UTF-16 surrogate low signature found without a
: leading surrogate high.
- : 36 A UTF-16 surrogate high signature found without two
: bytes for the surrogate low halfword.
- : 40 A UTF-16 surrogate high signature found without a
: trailing surrogate low.
- : 44 A UTF-32 code point is larger than the maximum
: allowed by Unicode.
- : 44 A UTF-32 code point is within the range assigned to
: UTF-16 surrogates.
- : **System Action:** The stage terminates with return code
: 1562.
-
- : **1563E Senary stream is incompatible with *word*.**
- : **Explanation:** When replacing a record, it must be unam-
: biguous which record to replace.
- : **System Action:** The stage terminates with return code
: 1563.
-
- : **1564E Beginning block number *number* larger than
: device capacity *number***
- : **System Action:** The stage terminates with return code
: 1564.
-
- : **1565E Ending block number *number* larger than device
: capacity *number***
- : **System Action:** The stage terminates with return code
: 1565.
-
- : **1566E Record size (*number* blocks) does not agree with
: block count *number* in record**
- : **System Action:** The stage terminates with return code
: 1566.
-
- : **1567E Beginning block *number* is greater than ending
: block *number***
- : **Explanation:** The third operand is larger than the fourth.
- : **System Action:** The stage terminates with return code
: 1567.
-
- : **1568E Block *number* before first writable block**
- : **System Action:** The stage terminates with return code
: 1568.
-
- : **1569E Block *number* after last writable block**
- : **System Action:** The stage terminates with return code
: 1569.
-
- : **1570E Mode *word* does not refer to an SFS directory**
- : **System Action:** The stage terminates with return code
: 1570.
-
- : **1571E Self-defining is too long (*number* bits): *string***
- : **System Action:** The stage terminates with return code
: 1571.
-
- : **1572E Needle cannot be empty**
- : **Explanation:** the second argument to SUBSTITUTE is a null
: string.
- : **System Action:** The stage terminates with return code
: 1572.
-
- : **1573E Argument *number* is required**
- : **Explanation:** An argument that is not optional was omitted.
- : **System Action:** The stage terminates with return code
: 1573.
-
- : **1574E Number is not an integer: *number***
- : **System Action:** The stage terminates with return code
: 1574.
-
- : **1575E First argument to D2C/D2X is negative, but
: second argument is omitted: *number***
- : **Explanation:** d2c() and d2x() with one argument support
: positive or zero only.
- : **System Action:** The stage terminates with return code
: 1575.
-
- : **1576E Stage is not running in a subroutine pipeline**
- : **Explanation:** *structure* with CALLER is not running in a
: subroutine pipeline. Thus, it has no caller and the caller
: scope does not exist.
- : **System Action:** The stage terminates with return code
: 1576.
-
- : **1577E No structures defined in caller**
- : **System Action:** The stage terminates with return code
: 1577.
-
- ! **1578E Cannot obtain lock for COMMAND stage (held by
: process *number* thread *number*)**
- ! **Explanation:** The lock to serialise *command* stages is held
: by another process or thread.
- ! **System Action:** The stage terminates with return code
: 1578.

! 1579W Cannot release lock for COMMAND stage: *hex*

! Explanation: The lock to serialise *command* stages could not be released. The contents of the lock are substituted. This is an error in *CMS Pipelines*. The error is ignored, but it may not be possible to run *command* or *cms* stages until *CMS Pipelines* is restarted.

! 1580E Do not convert numeric type member as if it were a string

! Explanation: One of the functions to convert from binary to numeric is applied to a member of a structure that is declared as having numeric type. The conversion is automatic; you should not need to do anything.

! User Response: Remove the erroneous function call.

! System Action: The stage terminates with return code 1580.

! 1581I Trap requested

! Explanation: Reserved for debugging the message trap.

! 1582I Trap issued the CMS command "*string*"

! Explanation: Informational message to indicate that an activated trap has sprung and a CMS command was issued through the subcommand interface.

! 1583I Trap dropped into CP Read

! Explanation: Informational message to indicate that an activated trap has sprung and that a CP console read was requested.

! 1584E Return code 70 renaming the file. Use >SFS instead

! Explanation: An erase and write operation is requested for a file. The file exists, so a utility file is written and renamed. The RENAME function fails with the undocumented return code 70, which is assumed to imply work unit trouble.

! User Response: Do not use the minidisk interface to files in the shared file system; use *>sfs* or specify a directory instead of a file mode.

! System Action: The stage terminates with return code 1584.

! 1585I Invoking CMS command *word* with header *hex* at *hex* in process *hex* thread *hex*

! Explanation: This is a debugging message.

! 1586I Return from CMS command *word*

! Explanation: This is a debugging message.

! 1587E No compression dictionary provided

! Explanation: The secondary input stream is connected, but no record is present that is not null.

! System Action: The stage terminates with return code 1587.

! 1588E Symbol translation and format-1 sibling descriptors are mutually exclusive

! Explanation: A number is specified for the symbol translation offset as well as the keyword FORMAT1. Results are documented for the hardware as unpredictable.

! System Action: The stage terminates with return code 1588.

! 1589E Dictionaries provided are *number*, but *number* is expected

! Explanation: A number is specified for the symbol translation offset. The specified number, when multiplied by 128 and increased by a quarter, is the expected size of the combined compression and symbol substitution dictionaries.

! System Action: The stage terminates with return code 1589.

! 1590E Dictionary size is not a multiple of 4096 (X'*number*')

! Explanation: No number is specified for the symbol translation offset or expansion is requested. The size of the first record that is not null on the secondary input stream is not a multiple of 4K.

! System Action: The stage terminates with return code 1590.

! 1591E Dictionary size is not a power of 2 (X'*number*')

! Explanation: No number is specified for the symbol translation offset or expansion is requested. The size of the first record that is not null on the secondary input stream is not a power of 2.

! System Action: The stage terminates with return code 1591.

! 1592E Dictionary size is too small

! Explanation: Format-1 sibling descriptors are specified, which effectively halves the size of the dictionary. As 4K are provided, this is too little.

! System Action: The stage terminates with return code 1592.

1593E Dictionary size is too large (*number*)

Explanation: The dictionary provided, possibly after halving its size for FORMAT1, is larger than 64K. This is larger than supported by the hardware.

System Action: The stage terminates with return code 1593.

1594I Issuing wait to operating system

Explanation: This is a debugging message.

1595E Prologue not recognised (*hex*)

Explanation: The stage has resolved to a save instruction that would indicate that the function is a C language program that has been processed by GCC for z/Linux, but the instructions that follow the save instruction are not as expected.

System Action: The stage terminates with return code 1595.

User Response: Contact your systems support staff.

System Programmer Response: The most likely cause is that the compiler has generated code that was not anticipated.

1596W Cannot erase original file renamed to *fileid*. Try SFS device driver instead

Explanation: A double rename and erase operation was attempted to replace a file on a mode letter. The replacement file has been written correctly, but the original file cannot be removed.

User Response: Check whether the work file still exists. If it does, and the mode letter is an accessed SFS directory, consider switching to use the native SFS device drivers, such as *>sfs* or specify a directory instead of a file mode.

1597E GLOBALV service not available

Explanation: Entry point for GLOBALV interface is not available.

System Programmer Response: The DMSGLOBE entry point is 0.

System Action: The stage terminates with return code 1597.

1598E Incorrect compression signature X'*hex*'

Explanation: The data can not be uncompressed by the selected protocol.

User Response: The first two bytes of the signature show the protocol used to compress the data. Find the corresponding protocol to expand the data. When the signature does not match any listed signatures, the data may have been encrypted or otherwise modified.

System Action: The stage terminates with return code 1598.

1599E Expansion failed after *number* bytes (*reason number*)

Explanation: The data is not correctly compressed.

User Response: Verify that the data was correctly transferred between systems and not truncated or padded incorrectly. While the mismatch is noticed at a specific location in the input data, there may have been errors earlier in the data that were not detected.

Contact your systems support staff. The reason code provides additional detail about the inconsistency in the data.

1 String code found where character is expected

2 Excessive chain of bytes

3 Potential loop in dictionary lookup

System Action: The stage terminates with return code 1599.

1600E FTP error: *string*

Explanation: The File Transfer Protocol server reports an error that prevents correct file transfer.

User Response: Investigate the error message from the FTP server and verify the specified URL. The TRACE option may be helpful to diagnose the problem.

System Action: The stage terminates with return code 1600.

1601E Error *number* parsing URL at "*string*"

Explanation: The argument is not a correctly formed Unified Resource Locator (URL). An error is found at or before the position reported.

1 No "scheme" specified in the URL.

2 Unsupported "scheme" in URL; supported schemes are "ftp" and "ftps".

3 Missing "hostname" in URL.

4 Incorrect qualifier; only "type=" is supported.

5 Incorrect type value; valid values are A, I, and D.

System Action: The stage terminates with return code 1601.

1602E Unable to open FTP data connection

System Action: The stage terminates with return code 1602.

1603E FTP processing error *number*

Explanation: The stage failed to process the response from the FTP server when requesting a port to connect the data channel.

1604I

- | 1 Port address section truncated.
 - | 2 Delimiters missing in 229 response.
 - | 3 Port number missing
 - | 4 IP address incomplete.
 - | 5 Port address in parentheses is missing.
- | **User Response:** Consider running stage with the TRACE option to diagnose the problem.
- | **System Action:** The stage terminates with return code 1603.

1604I FTP word "data"

| **Explanation:** The message displays diagnostic information produced as result of the TRACE option. The first word indicates the type of diagnostic information, the data shows the message exchanged.

| >> Command sent to the FTP Server.

| << Response received from the FTP Server. The responses start with a numeric code as documented in RFC 959 and later. Response codes in the "2xx" range indicate success, the "5xx" responses indicate failure.

| -- Additional diagnostic information.

Chapter 27. PIPMOD Command (*CMS Pipelines* only)

You do not need to understand PIPMOD to run normal pipelines; this chapter is for “master plumbers”.

The PIPE Bootstrap Module

The first PIPE command in a CMS session invokes a small module. The PIPE module establishes the pipeline environment in this way:

- It tests if the PIPMOD nucleus extension is installed. It could be in a shared segment which is already loaded; for example, by the system profile.
- It installs the DMSPIPE MODULE as the nucleus extension PIPMOD, if the nucleus extension is not already installed.
- It issues the command PIPMOD INSTALL to make the main module initialise itself. PIPMOD then installs a PIPE nucleus extension to service future PIPE commands.
- It reissues the PIPE command to let it be processed by the newly installed nucleus extension.

CMS Pipelines uses two nucleus extensions, because PIPMOD is reentrant and refreshable and is thus loaded into system storage to be protected from user programming errors. The PIPE command, however, can potentially run user programs and therefore uses user storage for work areas. Running the PIPE command as a user command also means that CMS will perform ABEND recovery in the event of an error.

You can drop the PIPE nucleus extension at any time; it will be installed again by the next PIPE command.

Red Neon!

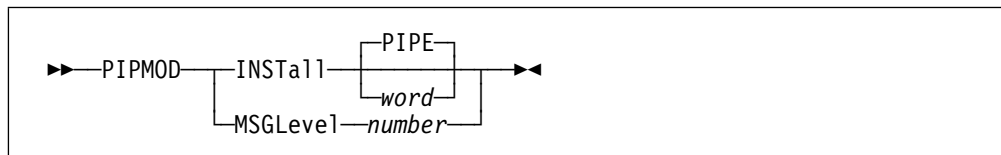
Do not drop the PIPMOD nucleus extension from a pipeline stage (for example, *cms*). *CMS Pipelines* issues message 107 when its service entry point is called as a result of the NUCXDROP command, but it is unable to prevent the storage occupied by the module from being released. An ABEND is most probable when control returns to *CMS Pipelines*.

The PIPMOD Nucleus Extension

The main pipeline module is installed as the nucleus extension PIPMOD. It can be invoked in three ways:

- As a normal CMS command. The PIPMOD command is used to set permanent options for *CMS Pipelines*. (It was used previously for the functions performed by the *query* and *help* services; use the PIPE command to invoke these services.)
- As an immediate command. Refer to “PIPMOD Immediate Commands” on page 865.
- As a program (CMSCALL TYPE=PROGRAM). This interface is used to install and remove filter packages. The details of this interface are unspecified and may change at any time; they are implemented by the filter package glue modules PIPNXF (Program Offering), DMSPPF (VM/ESA), and FPLNXF and FPLXG (runtime library and z/VM).

Setting Permanent Pipeline Options



INSTALL Create the PIPE nucleus extension (if it does not already exist) and install filter packages that are not already installed. If a word is specified, it is used instead of PIPE as the name of the command that runs a pipeline specification.

MSGLEVEL Set the rightmost halfword of the pipeline message level to the number that follows. The number is **decimal** (unlike the option on *runpipe*). The PIPMOD MSGLEVEL command can set all of the rightmost sixteen bits of the message level (unlike the global or local option MSGLEVEL, which is masked by X'17FF'). The individual bits are explained below.

The Message Level

Figure 406 shows the bits defined in the rightmost halfword of the pipeline message level.

Figure 406 (Page 1 of 2). Bits of the Message Level

Hex	Dec	Description
X'8000'		Undefined.
X'4000'		Undefined.
X'2000'	8,192	Account for time spent in stages and pipeline services. A message showing the amount of time spent in the stage is issued as each stage terminates. At the completion of a pipeline set, messages are issued to show the time spent in the scanner, the dispatcher, and various other internal services. Use of this facility can add substantial overhead if the underlying hardware does not support timing assists.
X'1000'	4,096	Check dispatcher entries thoroughly. Whenever it is called, the pipeline dispatcher ensures that it is running in user key (key X'E0') and that it is enabled for interrupts. This facility is used when testing <i>CMS Pipelines</i> . Use of this facility will add some overhead to the dispatcher path.
X'0800'	2,048	Account for storage allocated. When this facility is enabled, the <i>CMS Pipelines</i> storage manager keeps track of storage allocated to individual stages and to common services. It verifies that: <ul style="list-style-type: none"> • All storage allocated by a stage is released before the stage terminates and that all storage allocated by a pipeline set is released before the pipeline set terminates. • The same amount of storage is released as was allocated. • The doubleword following an allocated area is not overwritten. When a check fails, a message may be issued or the storage manager may force an assert failure.

Figure 406 (Page 2 of 2). Bits of the Message Level

Hex	Dec	Description
X'0400'	1,024	Enable additional debugging messages. Some built-in programs issue additional messages or write additional output records when this bit is enabled. The format of this information is unspecified.
X'0200'	512	Trace storage management. Messages are issued as storage is allocated and released. The format of these messages is unspecified. The bit for 2048 must also be turned on to obtain a reliable trace of storage allocation.
X'0100'	256	Stack messages. <i>CMS Pipelines</i> messages are queued on the program stack rather than issued (written to the terminal). This facility is obsolete. Use <i>runpipe</i> to obtain messages issued by a pipeline set.
X'0080'	128	Reserved for debugging of individual stages. Do not enable this bit.
X'0040'	64	Reserved for debugging of individual stages. Do not enable this bit.
X'0020'	32	Reserved for debugging of individual stages. Do not enable this bit.
X'0010'	16	Reserved for debugging of individual stages. Do not enable this bit.
X'0008'	8	Undefined.
X'0004'	4	Issue message 4 after an error message has been issued.
X'0002'	2	Issue message 2 after an error message has been issued while processing a pipeline command.
X'0001'	1	Issue message 1 after an error message has been issued.

The default message level is 15 (X'000F').

PIPMOD Immediate Commands


The following immediate commands are available:

ACTIVE	Show the active stage.
STOP	Terminate all stages that are waiting on an asynchronous event or the currently active stage.
WHERE	Display the addresses of the active pipeline control blocks.

CMS does not allow immediate commands to issue error messages. Thus, PIPMOD issues diagnostic messages and responses through CP messages to the virtual machine.

PIPMOD Command

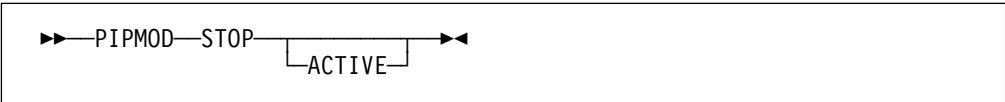
ACTIVE—Show the Active Stage



▶▶—PIPMOD—ACTIVE—◀◀

In general the currently active stage will change as the dispatcher manages the flow of work in the pipeline.

STOP—Terminating Stages that Wait Forever



▶▶—PIPMOD—STOP—
└—ACTIVE—┘◀◀


When ACTIVE is omitted all stages that are waiting for an asynchronous event are signalled that they should terminate. It can be used to stop a stage that is waiting for an event that will never occur. Note that CMS services are in general synchronous; the immediate command PIPMOD will not terminate a stage such as *disk*. Note that PIPMOD STOP does not terminate a stage that is in a loop unless it calls the dispatcher (use HI for REXX programs or HX). See “Device Drivers that Wait for External Events” on page 251 for a list of the built-in programs that can potentially wait on asynchronous events.

PIPMOD STOP ACTIVE signals the currently running stage to stop.

A stopable stage (sometimes called summarily stopable) is a stage that has specified that it uses only resources that the dispatcher knows how to release. It is not documented which stages are stopable (and this attribute may indeed change for a stage over time), but most filters are, including *spec*.

When the stage is terminated because the dispatcher sees the flag to stop, the stage is resumed with return code -4092. When the running stage is stopable and the I/O old PSW does not indicate that control is in the pipeline dispatcher, the stage will be forcefully terminated by making it return to the dispatcher with return code -4091 immediately when DMSITI returns to the interrupted program.

WHERE—Show Addresses of Pipeline Control Blocks



▶▶—PIPMOD—WHERE—◀◀

Chapter 28. Configuring *CMS Pipelines*

The “Field Test Version” of *CMS Pipelines* and the implementation shipped with z/VM have diverged in some respects over time. For example, the way a file is replaced is different when the file is in an SFS directory that is accessed as a mode letter.

As of level 1.1.10/0015, these differences are no longer fixed in the code; instead, the behaviour is specified by *pipeline configuration variables*.

There are two types of configuration variables, keyword variables and value variables. Keyword variables must be set to one of the supported keywords; a value variable specifies a word of up to eight characters to be used in some context. Case is ignored in the names of configuration variables; value variables are folded to upper case.

A default is determined by this hierarchy, where the first item has highest priority:

1. It can be set explicitly by the *configure* built-in program.
2. It can be stored in the system variable repository (a GLOBALV variable on CMS).
3. It is inferred from a default style (see below).

For each variable that has not been set explicitly, *CMS Pipelines* reads the variable from the system variable repository the first time it needs to inspect a particular variable; it saves the value internally from then on.

Default Styles

The default style specifies the default for configuration variables that you have not set explicitly and for which there is no variable in the system variable repository. You can select one of three default styles by setting the configuration variable STYLE:

DMS	This style sets the defaults to the values associated with the behaviour in z/VM. This style is the default for the DMSPIPE module shipped with z/VM.
PIP	This style sets the defaults to the values associated with the field test version of <i>CMS Pipelines</i> prior to 1.1.10/0015. This style is the default for the “runtime distribution” and for the module available from the VMTOOLS tools disk internally in IBM.
FPL	This style sets the defaults to a mixture of the two previous styles. It represents the recommended choice for each variable.

CMS Considerations

On CMS, the pipeline variables are stored by default within the GLOBALV group FPL. The group to use for subsequent queries can be changed by

```
pipe literal group MYPIPE | configure
```

The variables may be set using any method, such as

- `globalv select fpl setp diskreplace copy`
- Setting the desired variables in the file INITIAL GLOBALV.

Configuration Variables

Configuration Variables

The following sections contain an alphabetical list of *CMS Pipelines*'s configuration variables. The name of the variable is used for the section heading. The valid values for keyword variables are described in tables. The three columns of each table contain:

1. The value or keyword.
2. The style in which it is the default, if any.
3. A description.

Diskreplace

This keyword variable controls how *>mdsk* replaces a file in an SFS directory that is accessed with a mode letter. After the data are written to a temporary file, *CMS Pipelines* replaces either the file or its contents. The acceptable values are:

Copy	DMS	Use the DMSFILEC callable service to make the SFS server replace the previous contents of the output file with the contents of the temporary file. This retains the characteristics of the file at the expense of additional I/O in the SFS server virtual machine.
Replace	PIP FPL	Use the copy/erase method of replacing the old file with the new one. This reduces the load on the SFS server; but because the file is replaced, all authorisations are lost and the file creation date is changed.

Notes:

1. Since level 1.1.9, *CMS Pipelines* has been able to replace files directly in a SFS directory (CMS9 required). Specify the directory where the file resides instead of the mode letter. When a directory is specified, *>* uses native CSL routines to replace the file and this configuration variable becomes irrelevant.

Disktempfiletype

! This keyword variable specifies the file type used by *>mdsk* when it replaces a file on a
! minidisk or a file in SFS that is accessed as a mode letter.

TOD		Use the contents of the time-of-day clock as file name and file type. The sixty-four bits are unpacked to sixteen bytes printable hexadecimal. The file name and file type will be unique across a system, but not necessarily across a collection or an AVS network.
CMSUT1	DMS PIP FPL	Use the file type CMSUT1 and a file name related to the particular stage; this will be unique within the virtual machine.
USERID		Use the user ID as reported by diagnose 0 for the file type. Use a file name that is related to the particular stage. Thus, for all virtual machines that use the USERID keyword, the temporary file will be unique, but also easily found.

Notes:

1. Since level 1.1.9, *CMS Pipelines* has been able to replace files directly in a SFS directory (CMS9 required). Specify the directory where the file resides instead of the mode letter. When a directory is specified, > uses native CSL routines to replace the file and this configuration variable becomes irrelevant.

Group

The value of this variable specifies the GLOBALV group where configuration variables are stored. The default group is FPL in all styles; it can be changed only by the *configure* built-in program (clearly, the group cannot be specified by a variable within itself).

Repository

The value of this variable specifies the message repository to use for messages. A single hyphen (-) means that no message repository is used; *CMS Pipelines* then issues built-in messages.

Specify three letters to use a repository. The corresponding message repository is xxxUMEy TEXT, where xxx represents the three letters you specified; y is a code that represent the default language for your session.

-	PIP FPL	Do not use a message repository. Use the message texts that are stored within the pipeline module. The module prefix used is the same as the default style.
FPL	DMS	Use the default message repository for z/VM.

SQLpgmname

The value of this variable specifies the program name to use by the *sql* stage. The program name must match the value specified by the PREP= operand of the SQLPREP command that generated the access module.

DMSPQI	DMS FPL	The z/VM default.
PIPSQI	PIP	The original name.

SQLpgmowner

The value of this variable specifies the program owner to use by the *sql* stage. The program owner must match the SQL user ID that issued the SQLPREP command that generated the access module.

DMSPIPE	DMS FPL	The z/VM default.
5785RAC	PIP	The original name.

Notes:

1. The user ID must begin with a letter when DB2 is used.

Configuration Variables

Stallaction

This keyword variable controls the behaviour when the pipeline dispatcher determines that the pipeline is stalled. The original implementation issued message 29 and a message for each stage in the pipeline set; it then appended a formatted dump of the control block structure to a disk file.

JEREMY	FPL	Issue message 29 and append a readable summary of the status of each stage to the file specified by the STALLFILETYPE configuration variable.
QUIET		Issue message 29 only. Do not append to file.
STATUS		Issue message 29 and a message for each stage to show the state it is in. Do not append to file.
STATUSDUMP	PIP DMS	Issue message 29; issue a message for each stage to show the state it is in; and append a formatted dump of the control block structure to the file specified by the STALLFILETYPE configuration variable.

Stallfiletype

The value of this variable controls the file type used when appending the status of the stages in a pipeline set to a dump file. The STALLFILETYPE configuration variable is of interest only when the STALLACTION configuration variable is set to JEREMY or to STATUSDUMP.

When the last character of the STALLFILETYPE configuration variable is an asterisk (*), the characters before the asterisk are used as they are; the remaining characters up to a length of eight are inserted as a number, starting with zeros. *CMS Pipelines* loops incrementing this number until it finds a file type for which there is no file. If all files exist, the dump is appended to the last file found.

LISTING	PIP FPL	Append all dumps to a single file, PIPDUMP LISTING.
LIST00*	DMS	Write the dump to the first file that does not exist with a file type as specified. When all hundred files exist, append the dump to the file PIPDUMP LIST0099.

Style

This keyword variable controls the defaults for other variables. There are three default styles:

DMS	Use the behaviour of VM/ESA and z/VM.
PIP	Use the behaviour of the original Program Offering and the subsequent “field test versions”.
FPL	Use the recommended set.

For compatibility, the default for the Style configuration variable is DMS for z/VM; it is PIP for the field test version.

The style also governs other actions, which cannot be controlled through individual variables:

Notes:

- ! 1. The way *help* works is governed by the active style. For PIP and FPL, it behaves as *ahelp*, which displays information from PIPELINE HELPLIB (if it is available); for DMS, ! it issues the CMS HELP command to display a standard z/VM help file.
- 2. The application ID in the message prefix is controlled by the default style when the repository value is set to a hyphen. In the PIP style, the application ID is set to PIP; in the other two styles it is set to FPL.

Installation-wide Customisation (CMS)

- ! A z/VM installation can force a particular default setting by passing the appropriate record to *configure* in the system profile after the pipeline segment has been loaded.

Chapter 29. Diagnosis

This chapter contains information that may help you diagnose problems with *CMS Pipelines* or CMS.

Determining and Terminating the Currently Running Stage

You can determine the currently running stage through an immediate command; the procedure is different on virtual machine and z/OS.

VM

The PIPMOD immediate command support the options ACTIVE and STOP ACTIVE. As immediate commands are not allowed to perform I/O, the command response is via CP messages.

The commands are applied to all threads. This discussion assumes that only the main thread is running a pipeline; that is, we ignore CMS multitasking.

PIPMOD ACTIVE shows the currently running stage.

PIPMOD STOP ACTIVE sets a flag for the dispatcher to terminate the stage, but whether the dispatcher sees this flag is another matter.

Refer to “PIPMOD Immediate Commands” on page 865 for a discussion of these two immediate commands.

A stage that is not stopable and in a loop will not call the dispatcher and therefore it cannot be terminated. CMS command HX is still the only option in this case.

Also note that CMS will interpret the PIPMOD command as a normal command when entered at the Ready prompt. Neither ACTIVE nor STOP are valid PIPMOD subcommands.

Figure 407. Sample Use of PIPMOD Immediate Commands

```

: pipe literal|dup *|count lines|cons
: pipmod active
: 12:55:20 * MSG FROM JOHN : PIPMOD1479I Running: dup *
: 12:55:20 * MSG FROM JOHN : PIPMSG003I ... Issued from stage 2 of pipeline 1
: 12:55:20 * MSG FROM JOHN : PIPMSG001I ... Running "dup *"
: pipmod stop active
: 12:55:27 * MSG FROM JOHN : PIPMOD1479I Running: count lines
: 12:55:27 * MSG FROM JOHN : PIPMSG003I ... Issued from stage 3 of pipeline 1
: 12:55:27 * MSG FROM JOHN : PIPMSG001I ... Running "count lines"
: 12:55:27 * MSG FROM JOHN : PIPMOD1482I Stage is in the dispatcher; likely to stop on th
: Ready(-4092); T=11.05/11.11 12:55:27

: term linend off
: Ready; T=0.01/0.01 13:13:04
: pipe literal | spec while #0=0 do set #1+=1 done print #1 1 | cons
: pipmod active
: 13:13:13 * MSG FROM JOHN : PIPMOD1479I Running: spec while #0=0 do set #1+=1 done print
: 13:13:13 * MSG FROM JOHN : PIPMSG003I ... Issued from stage 2 of pipeline 1
: 13:13:13 * MSG FROM JOHN : PIPMSG001I ... Running "spec while #0=0 do set #1+=1 done pr
: 13:13:13 * MSG FROM JOHN : PIPMOD1480I In procedure COMPARE
: pipmod stop active
: 13:13:21 * MSG FROM JOHN : PIPMOD1479I Running: spec while #0=0 do set #1+=1 done print
: 13:13:21 * MSG FROM JOHN : PIPMSG003I ... Issued from stage 2 of pipeline 1
: 13:13:21 * MSG FROM JOHN : PIPMSG001I ... Running "spec while #0=0 do set #1+=1 done pr
: 13:13:21 * MSG FROM JOHN : PIPMOD1486I Stage is flagged to stop. Forcing exit from FPL
: Ready(-4091); T=11.52/11.58 13:13:21

```

Traps

A trap facility is implemented for *CMS Pipelines* to allow the user to specify CP commands to issue when a particular message is issued.

The facility is enabled for messages FPLDSK124E and FPLDSK129E. There is no user interface to enable other messages, though it would be possible to write a REXX program to enable other messages in the field, should this be necessary.

When an enabled message has been issued, the global variable corresponding to the module, message, and severity (for example, DSK124E) is fetched from the global variable group FPLTRAP and inspected. If the contents of the variable are not blank, its value is stripped of leading blanks and passed directly to CP with diagnose 8 and then logged with message 1453. The return code is ignored.

The most useful command to issue by these means is expected to be the CP command VMDUMP, but you could also display storage or send a message to someone instead (or given sufficient privileges, shut down the system).

The usual rules apply: The command is limited to 240 bytes; it should be upper case unless for strings that you wish to be mixed case; line end characters (X'15') separate multiple commands.

It is deliberate that the command is issued directly to CP; this makes the current register set easy to find (registers 0 through 12 are the ones of the program that issued the message);

Traps

: had a CMS command been issued instead, it would be tedious to find the registers (though
: clearly possible given sufficient stamina).

: Assuming you create a VMDUMP SPOOL file, you can use DUMpload to read it in, then
: access MAINT 193 and inspect it with VMDUMPTL. The *CMS Pipelines* home page contains
: a sample VM Dump Tool macro, FPLDSK VMDT, to extract information from the dump file
: for the two enabled messages. Refer to:

: <http://vm.marist.edu/%7epipeline/FPLDSK.vmdt>

: For a task-oriented explanation of what to do with a dump, refer to:

: <http://vm.marist.edu/%7epipeline/traps.html>

Part 5. Appendices

This part of the book contains miscellaneous information.

Appendix A, “Summary of Built-in Programs” contains the synopses for the built-in programs, ordered by keywords; it can be useful when looking for a program to perform a particular function.

Appendix B, “Messages, Sorted by Text” contains a reference to message numbers in the order of their text.

Appendix C, “Implementing CMS Commands as Stages in a Pipeline” describes how some CMS commands can be formulated as pipeline specifications; it shows device drivers and filters that can be used to accomplish the equivalent function of a CMS command. This is intended for the experienced CMS user who wishes to perform *almost* the function of a CMS command. Given how a CMS command is implemented with *CMS Pipelines*, it is easy to tune the pipeline specification to a slightly different operation.

Appendix D, “Running Multiple Versions of *CMS Pipelines* Concurrently” describes how *CMS Pipelines* initialises itself when the first PIPE command is issued in a CMS session. It also explains how different versions of *CMS Pipelines* can be active concurrently in a virtual machine.

Appendix E, “Generating and Using Filter Packages with *CMS Pipelines*” describes filter packages and explains how to generate them.

Appendix F, “Pipeline Compatibility and Portability between CMS and TSO” describes compatibility and portability between the CMS and z/OS environments.

Appendix G, “Format of Output Records from *runpipe* EVENTS” contains information useful for the authors of PIPEDEMO, RITA, and others who process the detailed trace of *CMS Pipelines* operation that is produced by *runpipe* EVENTS.

Appendix A. Summary of Built-in Programs

This appendix contains a keyword out of context listing of the synopses of the built-in programs. The first column is a keyword, the second column has the name of the stage, and the remainder of the line is the synopsis for the program.

*ACCOUNT	<i>starsys</i>	Write Lines from a Two-way CP System Service
*LOGREC	<i>starsys</i>	Write Lines from a Two-way CP System Service
*MONITOR	<i>starmon</i>	Write Records from the *MONITOR System Service
*MSG	<i>starmsg</i>	Write Lines from a CP System Service
*MSGALL	<i>starmsg</i>	Write Lines from a CP System Service
*SYMPTOM	<i>starsys</i>	Write Lines from a Two-way CP System Service
abbreviation	<i>abbrev</i>	Select Records that Contain an Abbreviation of a Word in the First Positions
access list	<i>alserv</i>	Manage the Virtual Machine's Access List
ACI	<i>acigroup</i>	Write ACI Group for Users
active	<i>aftfst</i>	Write Information about Open Files
address	<i>ip2socka</i>	Build sockaddr_in Structure
address	<i>socka2ip</i>	Format sockaddr_in Structure
address	<i>3277bfra</i>	Convert a 3270 Buffer Address Between Representations
AES	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher
AFT	<i>aftfst</i>	Write Information about Open Files
after	<i>append</i>	Put Output from a Device Driver after Data on the Primary Input Stream
after	<i>preface</i>	Put Output from a Device Driver before Data on the Primary Input Stream
aggregate	<i>aggrc</i>	Compute Aggregate Return Code
ALET	<i>alserv</i>	Manage the Virtual Machine's Access List
align	<i>scm</i>	Align REXX Comments
and	<i>combine</i>	Combine Data from a Run of Records
APL	<i>apldecode</i>	Process Graphic Escape Sequences
APL	<i>aplencode</i>	Generate Graphic Escape Sequences
append	<i>>>mdsk</i>	Append to or Create a CMS File on a Mode
append	<i>>>sfs</i>	Append to or Create an SFS File
append	<i>>>sfsslow</i>	Append to or Create an SFS File
append	<i>append</i>	Put Output from a Device Driver after Data on the Primary Input Stream
append	<i>diskfast</i>	Read, Create, or Append to a File
append	<i>diskslow</i>	Read, Create, or Append to a File
append	<i>diskupdate</i>	Replace Records in a File
append	<i>fanin</i>	Concatenate Streams
append	<i>join</i>	Join Records
append	<i>joincont</i>	Join Continuation Lines
append	<i>mdskfast</i>	Read, Create, or Append to a CMS File on a Mode
append	<i>mdskslow</i>	Read, Append to, or Create a CMS File on a Mode
append	<i>mdskupdate</i>	Replace Records in a File on a Mode
append	<i>preface</i>	Put Output from a Device Driver before Data on the Primary Input Stream
arrange	<i>spec</i>	Rearrange Contents of Records
ASA	<i>asatomc</i>	Convert ASA Carriage Control to CCW Operation Codes
ASA	<i>mctoasa</i>	Convert CCW Operation Codes to ASA Carriage Control
ASCII	<i>qpdecode</i>	Decode to Quoted-printable Format
ASCII	<i>qpencode</i>	Encode to Quoted-printable Format

ASCII	<i>urldeblock</i>	Process Universal Resource Locator
assembler	<i>asmcont</i>	Join Multiline Assembler Statements
assembler	<i>asmfind</i>	Select Statements from an Assembler File as XEDIT Find
assembler	<i>asmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
assembler	<i>asmxpnd</i>	Expand Joined Assembler Statements
assembler	<i>hlasm</i>	Interface to High Level Assembler
assembler	<i>hlasmerr</i>	Extract Assembler Error Messages from the SYSADATA File
assembler	<i>strasmfind</i>	Select Statements from an Assembler File as XEDIT Find
assembler	<i>strasmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
ATTN	<i>stack</i>	Read or Write the Program Stack
attribute	<i>xab</i>	Read or Write External Attribute Buffers
backwards	<i>diskback</i>	Read a File Backwards
backwards	<i>mdskback</i>	Read a CMS File from a Mode Backwards
backwards	<i>noeofback</i>	Pass Records and Ignore End-of-file on Output
backwards	<i>sfsback</i>	Read an SFS File Backwards
base-64	<i>64decode</i>	Decode MIME Base-64 Format
base-64	<i>64encode</i>	Encode to MIME Base-64 Format
between	<i>between</i>	Select Records Between Labels
between	<i>inside</i>	Select Records between Labels
between	<i>notininside</i>	Select Records Not between Labels
between	<i>outside</i>	Select Records Not between Labels
BFS	<i>filedescriptor</i>	Read or Write an OpenExtensions File that Is Already Open
bits	<i>vchar</i>	Recode Characters to Different Length
block	<i>block</i>	Block to an External Format
block	<i>deblock</i>	Deblock External Data Formats
block	<i>fblock</i>	Block Data, Spanning Input Records
blocking	<i>addrdw</i>	Prefix Record Descriptor Word to Records
Blowfish	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher
browse	<i>browse</i>	Display Data on a 3270 Terminal
buffer	<i>buffer</i>	Buffer Records
buffer	<i>dfsrt</i>	Interface to DFSORT/CMS
buffer	<i>elastic</i>	Buffer Sufficient Records to Prevent Stall
buffer	<i>instore</i>	Load the File into a storage Buffer
buffer	<i>outstore</i>	Unload a File from a storage Buffer
buffer	<i>sort</i>	Order Records
buffer	<i>xab</i>	Read or Write External Attribute Buffers
buffer	<i>3277bfra</i>	Convert a 3270 Buffer Address Between Representations
byte	<i>parcel</i>	Parcel Input Stream Into Records
byte files	<i><oe</i>	Read an OpenExtensions Text File
byte files	<i>>>oe</i>	Append to or Create an OpenExtensions Text File
byte files	<i>>oe</i>	Replace or Create an OpenExtensions Text File
byte files	<i>filedescriptor</i>	Read or Write an OpenExtensions File that Is Already Open
byte files	<i>hfs</i>	Read or Append File in the Hierarchical File System
byte files	<i>hfsdirectory</i>	Read Contents of a Directory in a Hierarchical File System
byte files	<i>hfsquery</i>	Write Information Obtained from OpenExtensions into the Pipeline
byte files	<i>hfsreplace</i>	Replace the Contents of a File in the Hierarchical File System
byte files	<i>hfsstate</i>	Obtain Information about Files in the Hierarchical File System
byte files	<i>hfsxecute</i>	Issue OpenExtensions Requests
byte stream	<i>block</i>	Block to an External Format
byte stream	<i>deblock</i>	Deblock External Data Formats
bytes	<i>count</i>	Count Lines, Blank-delimited Words, and Bytes

Stage Selection Guide

cards	<i>punch</i>	Punch Cards
cards	<i>reader</i>	Read from a Virtual Card Reader
cards	<i>uro</i>	Write Unit Record Output
carriage	<i>asatomc</i>	Convert ASA Carriage Control to CCW Operation Codes
carriage	<i>buildscr</i>	Build a 3270 Data Stream
carriage	<i>c14to38</i>	Combine Overstruck Characters to Single Code Point
carriage	<i>mctoasa</i>	Convert CCW Operation Codes to ASA Carriage Control
carriage	<i>overstr</i>	Process Overstruck Lines
carriage	<i>printmc</i>	Print Lines
case	<i>casei</i>	Run Selection Stage in Case Insensitive Manner
case	<i>zone</i>	Run Selection Stage on Subset of Input Record
CAT	<i>optcdj</i>	Generate Table Reference Character (TRC)
catenate	<i>join</i>	Join Records
catenate	<i>joincont</i>	Join Continuation Lines
CCW	<i>asatomc</i>	Convert ASA Carriage Control to CCW Operation Codes
CCW	<i>mctoasa</i>	Convert CCW Operation Codes to ASA Carriage Control
century	<i>dateconvert</i>	Convert Date Formats
CGI	<i>urldeblock</i>	Process Universal Resource Locator
change	<i>change</i>	Substitute Contents of Records
character	<i>vchar</i>	Recode Characters to Different Length
CHARS	<i>optcdj</i>	Generate Table Reference Character (TRC)
checksum	<i>tcpcksum</i>	Compute One's complement Checksum of a Message
chop	<i>chop</i>	Truncate the Record
cipher	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher
CKD	<i>ckddeblock</i>	Deblock Track Data Record
client	<i>vmclient</i>	Send VMCF Requests
CLIST	<i>rexxvars</i>	Retrieve Variables from a REXX or CLIST Variable Pool
CLIST	<i>stem</i>	Retrieve or Set Variables in a REXX or CLIST Variable Pool
CLIST	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
CLIST	<i>varload</i>	Set Variables in a REXX or CLIST Variable Pool
CLIST	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
CMS	<i><mnsk</i>	Read a CMS File from a Mode
CMS	<i><sfs</i>	Read an SFS File
CMS	<i><sfsslow</i>	Read an SFS File
CMS	<i>>>mnsk</i>	Append to or Create a CMS File on a Mode
CMS	<i>>>sfs</i>	Append to or Create an SFS File
CMS	<i>>>sfsslow</i>	Append to or Create an SFS File
CMS	<i>>mnsk</i>	Replace or Create a CMS File on a Mode
CMS	<i>>sfs</i>	Replace or Create an SFS File
CMS	<i>cms</i>	Issue CMS Commands, Write Response to Pipeline
CMS	<i>command</i>	Issue CMS Commands, Write Response to Pipeline
CMS	<i>diskfast</i>	Read, Create, or Append to a File
CMS	<i>diskslow</i>	Read, Create, or Append to a File
CMS	<i>filetoken</i>	Read or Write an SFS File That is Already Open
CMS	<i>mnskfast</i>	Read, Create, or Append to a CMS File on a Mode
CMS	<i>sfsback</i>	Read an SFS File Backwards
CMS	<i>sfsrandom</i>	Random Access an SFS File
CMS	<i>sfsupdate</i>	Replace Records in an SFS File
CMS	<i>subcom</i>	Issue Commands to a Subcommand Environment
CMSCALL	<i>cms</i>	Issue CMS Commands, Write Response to Pipeline
code	<i>aggrc</i>	Compute Aggregate Return Code
code	<i>crc</i>	Compute Cyclic Redundancy Code
codes	<i>sqlcodes</i>	Write the last 11 SQL Codes Received
collate	<i>collate</i>	Collate Streams

collect	<i>fanin</i>	Concatenate Streams
collect	<i>faninany</i>	Copy Records from Whichever Input Stream Has One
collect	<i>fanintwo</i>	Pass Records to Primary Output Stream
columns	<i>snake</i>	Build Multicolumn Page Layout
combine	<i>collate</i>	Collate Streams
command	<i>cms</i>	Issue CMS Commands, Write Response to Pipeline
command	<i>command</i>	Issue TSO Commands
command	<i>command</i>	Issue CMS Commands, Write Response to Pipeline
command	<i>cp</i>	Issue CP Commands, Write Response to Pipeline
command	<i>immcmd</i>	Write the Argument String from Immediate Commands
command	<i>mqsc</i>	Issue Commands to a WebSphere MQ Queue Manager
command	<i>tso</i>	Issue TSO Commands, Write Response to Pipeline
commands	<i>pipcmd</i>	Issue Pipeline Commands
comments	<i>scm</i>	Align REXX Comments
compare	<i>pick</i>	Select Lines that Satisfy a Relation
compiler	<i>polish</i>	Reverse Polish Expression Parser
conditional	<i>if</i>	Process Records Conditionally
configuration	<i>configure</i>	Set and Query <i>CMS Pipelines</i> Configuration Variables
console	<i>fullscrq</i>	Write 3270 Device Characteristics
console	<i>stack</i>	Read or Write the Program Stack
continuation	<i>asmcont</i>	Join Multiline Assembler Statements
continuation	<i>asmxpnd</i>	Expand Joined Assembler Statements
continuation	<i>joincont</i>	Join Continuation Lines
contract	<i>retab</i>	Replace Runs of Blanks with Tabulate Characters
control	<i>asatomc</i>	Convert ASA Carriage Control to CCW Operation Codes
control	<i>mctoasa</i>	Convert CCW Operation Codes to ASA Carriage Control
conversion	<i>dateconvert</i>	Convert Date Formats
copies	<i>buffer</i>	Buffer Records
copies	<i>duplicate</i>	Copy Records
copipe	<i>fitting</i>	Source or Sink for Copipe Data
copy	<i>copy</i>	Copy Records, Allowing for a One Record Delay
copy	<i>dam</i>	Pass Records Once Primed
copy	<i>noeofback</i>	Pass Records and Ignore End-of-file on Output
count	<i>count</i>	Count Lines, Blank-delimited Words, and Bytes
count	<i>sort</i>	Order Records
count	<i>unique</i>	Discard or Retain Duplicate Lines
CP	<i>acigroup</i>	Write ACI Group for Users
CP	<i>cp</i>	Issue CP Commands, Write Response to Pipeline
CP	<i>devinfo</i>	Write Device Information
CP	<i>starmon</i>	Write Records from the *MONITOR System Service
CP	<i>starmsg</i>	Write Lines from a CP System Service
CP	<i>starsys</i>	Write Lines from a Two-way CP System Service
CP	<i>tsi</i>	Store System Information
CRC	<i>crc</i>	Compute Cyclic Redundancy Code
create	<i>>>mnsk</i>	Append to or Create a CMS File on a Mode
create	<i>>mnsk</i>	Replace or Create a CMS File on a Mode
create	<i>>sfs</i>	Replace or Create an SFS File
create	<i>diskfast</i>	Read, Create, or Append to a File
create	<i>diskslow</i>	Read, Create, or Append to a File
create	<i>diskupdate</i>	Replace Records in a File
create	<i>mnskfast</i>	Read, Create, or Append to a CMS File on a Mode
create	<i>mnskslow</i>	Read, Append to, or Create a CMS File on a Mode
create	<i>mnskupdate</i>	Replace Records in a File on a Mode
cyclic	<i>crc</i>	Compute Cyclic Redundancy Code

Stage Selection Guide

data	<i>structure</i>	Manage Structure Definitions
data set	<i><mvs</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set
data set	<i>>>mvs</i>	Append to a Physical Sequential Data Set
data set	<i>>mvs</i>	Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set
data set	<i>iebcopy</i>	Process IEBCOPY Data Format
data set	<i>listcat</i>	Obtain Data Set Names
data set	<i>listdsi</i>	Obtain Information about Data Sets
data set	<i>listispf</i>	Read Directory of a Partitioned Data Set into the Pipeline
data set	<i>listpds</i>	Read Directory of a Partitioned Data Set into the Pipeline
data set	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
data set	<i>readpds</i>	Read Members from a Partitioned Data Set
data set	<i>state</i>	Verify that Data Set Exists
data set	<i>sysdsn</i>	Test whether Data Set Exists
data set	<i>writepds</i>	Store Members into a Partitioned Data Set
data space	<i>adrspc</i>	Manage Address Spaces
data space	<i>mapmdisk</i>	Map Minidisks Into Data spaces
database	<i>sql</i>	Interface to SQL
database	<i>sqlselect</i>	Query a Database and Format Result
datagram	<i>udp</i>	Read and Write an UDP Port
date	<i>dateconvert</i>	Convert Date Formats
date	<i>greg2sec</i>	Convert a Gregorian Timestamp to Second Since Epoch
date	<i>sec2greg</i>	Convert Seconds Since Epoch to Gregorian Timestamp
date	<i>timestamp</i>	Prefix the Date and Time to Records
DB2	<i>sql</i>	Interface to SQL
DCB	<i><mvs</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set
DCB	<i>>>mvs</i>	Append to a Physical Sequential Data Set
DCB	<i>>mvs</i>	Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set
DCB	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
DCB	<i>sysout</i>	Write System Output Data Set
deblock	<i>deblock</i>	Deblock External Data Formats
deblock	<i>fblock</i>	Block Data, Spanning Input Records
deblock	<i>iebcopy</i>	Process IEBCOPY Data Format
decode	<i>64decode</i>	Decode MIME Base-64 Format
decrypt	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher
delay	<i>beat</i>	Mark when Records Do not Arrive within Interval
delay	<i>copy</i>	Copy Records, Allowing for a One Record Delay
delay	<i>delay</i>	Suspend Stream
DES	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher
describe	<i>sql</i>	Interface to SQL
descriptor	<i>addrdw</i>	Prefix Record Descriptor Word to Records
destruct	<i>predselect</i>	Control Destructive Test of Records
device	<i>devinfo</i>	Write Device Information
device	<i>eofback</i>	Run an Output Device Driver and Propagate End-of-file Backwards
device	<i>fullscrq</i>	Write 3270 Device Characteristics
device	<i>fullscrs</i>	Format 3270 Device Characteristics
device	<i>waitdev</i>	Wait for an Interrupt from a Device
diagnose	<i>fullscrq</i>	Write 3270 Device Characteristics
diagnose E0	<i>trfread</i>	Read a Trace File
diagnose E4	<i>diage4</i>	Submit Diagnose E4 Requests
diagnose 14	<i>reader</i>	Read from a Virtual Card Reader
diagnose 58	<i>fullscr</i>	Full screen 3270 Write and Read to the Console or Dialed/Attached Screen
diagnose 8	<i>cp</i>	Issue CP Commands, Write Response to Pipeline
directory	<i>hfsdirectory</i>	Read Contents of a Directory in a Hierarchical File System
directory	<i>listispf</i>	Read Directory of a Partitioned Data Set into the Pipeline

directory	<i>listpds</i>	Read Directory of a Partitioned Data Set into the Pipeline
directory	<i>sfsdirectory</i>	List Files in an SFS Directory
discard	<i>chop</i>	Truncate the Record
discard	<i>drop</i>	Discard Records from the Beginning or the End of the File
discard	<i>hole</i>	Destroy Data
discard	<i>strip</i>	Remove Leading or Trailing Characters
disk	<i>>>mnsk</i>	Append to or Create a CMS File on a Mode
disk	<i>>>sfs</i>	Append to or Create an SFS File
disk	<i>>>sfsslow</i>	Append to or Create an SFS File
disk	<i>>mnsk</i>	Replace or Create a CMS File on a Mode
disk	<i>>sfs</i>	Replace or Create an SFS File
disk	<i>diskfast</i>	Read, Create, or Append to a File
disk	<i>diskslow</i>	Read, Create, or Append to a File
disk	<i>diskupdate</i>	Replace Records in a File
disk	<i>fbaread</i>	Read Blocks from a Fixed Block Architecture Drive
disk	<i>fbawrite</i>	Write Blocks to a Fixed Block Architecture Drive
disk	<i>mdiskblk</i>	Read or Write Minidisk Blocks
disk	<i>mnskback</i>	Read a CMS File from a Mode Backwards
disk	<i>mnskfast</i>	Read, Create, or Append to a CMS File on a Mode
disk	<i>mnskrandom</i>	Random Access a CMS File on a Mode
disk	<i>mnskslow</i>	Read, Append to, or Create a CMS File on a Mode
disk	<i>mnskupdate</i>	Replace Records in a File on a Mode
disk	<i>members</i>	Extract Members from a Partitioned Data Set
disk	<i>pdsdirect</i>	Write Directory Information from a CMS Simulated Partitioned Data Set
disk	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
disk	<i>sfsupdate</i>	Replace Records in an SFS File
display	<i>browse</i>	Display Data on a 3270 Terminal
display	<i>fullscr</i>	Full screen 3270 Write and Read to the Console or Dialed/Attached Screen
driver	<i><mnsk</i>	Read a CMS File from a Mode
driver	<i><sfs</i>	Read an SFS File
driver	<i><sfsslow</i>	Read an SFS File
driver	<i>>>mnsk</i>	Append to or Create a CMS File on a Mode
driver	<i>>>sfs</i>	Append to or Create an SFS File
driver	<i>>>sfsslow</i>	Append to or Create an SFS File
driver	<i>>mnsk</i>	Replace or Create a CMS File on a Mode
driver	<i>>sfs</i>	Replace or Create an SFS File
driver	<i>affst</i>	Write Information about Open Files
driver	<i>cms</i>	Issue CMS Commands, Write Response to Pipeline
driver	<i>command</i>	Issue TSO Commands
driver	<i>command</i>	Issue CMS Commands, Write Response to Pipeline
driver	<i>cp</i>	Issue CP Commands, Write Response to Pipeline
driver	<i>diskfast</i>	Read, Create, or Append to a File
driver	<i>diskslow</i>	Read, Create, or Append to a File
driver	<i>diskupdate</i>	Replace Records in a File
driver	<i>eofback</i>	Run an Output Device Driver and Propagate End-of-file Backwards
driver	<i>filetoken</i>	Read or Write an SFS File That is Already Open
driver	<i>fullscr</i>	Full screen 3270 Write and Read to the Console or Dialed/Attached Screen
driver	<i>immcmd</i>	Write the Argument String from Immediate Commands
driver	<i>literal</i>	Write the Argument String
driver	<i>mdiskblk</i>	Read or Write Minidisk Blocks
driver	<i>mnskback</i>	Read a CMS File from a Mode Backwards
driver	<i>mnskfast</i>	Read, Create, or Append to a CMS File on a Mode
driver	<i>mnskrandom</i>	Random Access a CMS File on a Mode
driver	<i>mnskslow</i>	Read, Append to, or Create a CMS File on a Mode

Stage Selection Guide

driver	<i>mdskupdate</i>	Replace Records in a File on a Mode
driver	<i>members</i>	Extract Members from a Partitioned Data Set
driver	<i>pdsdirect</i>	Write Directory Information from a CMS Simulated Partitioned Data Set
driver	<i>printmc</i>	Print Lines
driver	<i>punch</i>	Punch Cards
driver	<i>reader</i>	Read from a Virtual Card Reader
driver	<i>sfsback</i>	Read an SFS File Backwards
driver	<i>sfsrandom</i>	Random Access an SFS File
driver	<i>sfsupdate</i>	Replace Records in an SFS File
driver	<i>sql</i>	Interface to SQL
driver	<i>sqlselect</i>	Query a Database and Format Result
driver	<i>stack</i>	Read or Write the Program Stack
driver	<i>starmsg</i>	Write Lines from a CP System Service
driver	<i>starsys</i>	Write Lines from a Two-way CP System Service
driver	<i>state</i>	Verify that Data Set Exists
driver	<i>state</i>	Provide Information about CMS Files
driver	<i>statew</i>	Provide Information about Writable CMS Files
driver	<i>stem</i>	Retrieve or Set Variables in a REXX or CLIST Variable Pool
driver	<i>storage</i>	Read or Write Virtual Machine Storage
driver	<i>strliteral</i>	Write the Argument String
driver	<i>subcom</i>	Issue Commands to a Subcommand Environment
driver	<i>tape</i>	Read or Write Tapes
driver	<i>tso</i>	Issue TSO Commands, Write Response to Pipeline
driver	<i>uro</i>	Write Unit Record Output
driver	<i>var</i>	Retrieve or Set a Variable in a REXX or CLIST Variable Pool
driver	<i>vardrop</i>	Drop Variables in a REXX Variable Pool
driver	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
driver	<i>varload</i>	Set Variables in a REXX or CLIST Variable Pool
driver	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
driver	<i>xedit</i>	Read or Write a File in the XEDIT Ring
driver	<i>xmsg</i>	Issue XEDIT Messages
drop	<i>drop</i>	Discard Records from the Beginning or the End of the File
dump	<i>jeremy</i>	Write Pipeline Status to the Pipeline
duplicate	<i>duplicate</i>	Copy Records
EBCDIC	<i>qpdecode</i>	Decode to Quoted-printable Format
EBCDIC	<i>qpencode</i>	Encode to Quoted-printable Format
EBCDIC	<i>urldeblock</i>	Process Universal Resource Locator
ECKD	<i>ckddeblock</i>	Deblock Track Data Record
ECKD	<i>trackblock</i>	Build Track Record
ECKD	<i>trackdeblock</i>	Deblock Track
ECKD	<i>trackread</i>	Read Full Tracks from ECKD Device
ECKD	<i>tracksquish</i>	Squish Tracks
ECKD	<i>trackverify</i>	Verify Track Format
ECKD	<i>trackwrite</i>	Write Full Tracks to ECKD Device
ECKD	<i>trackxpan</i>	Unsquish Tracks
edit	<i>asmcont</i>	Join Multiline Assembler Statements
edit	<i>change</i>	Substitute Contents of Records
edit	<i>chop</i>	Truncate the Record
edit	<i>pad</i>	Expand Short Records
edit	<i>spec</i>	Rearrange Contents of Records
edit	<i>split</i>	Split Records Relative to a Target
edit	<i>strip</i>	Remove Leading or Trailing Characters
edit	<i>xlate</i>	Transliterate Contents of Records

elastic	<i>copy</i>	Copy Records, Allowing for a One Record Delay
EMSG	<i>emsg</i>	Issue Messages
encode	<i>64encode</i>	Encode to MIME Base-64 Format
encoding	<i>3277enc</i>	Write the 3277 6-bit Encoding Vector
encrypt	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher
end-of-file	<i>eofback</i>	Run an Output Device Driver and Propagate End-of-file Backwards
end-of-file	<i>noeofback</i>	Pass Records and Ignore End-of-file on Output
error	<i>emsg</i>	Issue Messages
error	<i>hlasmerr</i>	Extract Assembler Error Messages from the SYSADATA File
error	<i>sqlcodes</i>	Write the last 11 SQL Codes Received
escape	<i>escape</i>	Insert Escape Characters in the Record
event	<i>pause</i>	Signal a Pause Event
exclusive	<i>combine</i>	Combine Data from a Run of Records
EXECCOMM	<i>rexxvars</i>	Retrieve Variables from a REXX or CLIST Variable Pool
EXECCOMM	<i>stem</i>	Retrieve or Set Variables in a REXX or CLIST Variable Pool
EXECCOMM	<i>var</i>	Retrieve or Set a Variable in a REXX or CLIST Variable Pool
EXECCOMM	<i>vardrop</i>	Drop Variables in a REXX Variable Pool
EXECCOMM	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
EXECCOMM	<i>varload</i>	Set Variables in a REXX or CLIST Variable Pool
EXECCOMM	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
execution	<i>if</i>	Process Records Conditionally
expand	<i>asmxpnd</i>	Expand Joined Assembler Statements
expand	<i>untab</i>	Replace Tabulate Characters with Blanks
extend	<i>pad</i>	Expand Short Records
extension	<i>nuext</i>	Call a Nucleus Extension
FBA	<i>fbaread</i>	Read Blocks from a Fixed Block Architecture Drive
FBA	<i>fbawrite</i>	Write Blocks to a Fixed Block Architecture Drive
file	<i><</i>	Read a File
file	<i><mdsk</i>	Read a CMS File from a Mode
file	<i><mvs</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set
file	<i><oe</i>	Read an OpenExtensions Text File
file	<i><sfs</i>	Read an SFS File
file	<i><sfsslow</i>	Read an SFS File
file	<i>></i>	Replace or Create a File
file	<i>>></i>	Append to or Create a File
file	<i>>>mdsk</i>	Append to or Create a CMS File on a Mode
file	<i>>>mvs</i>	Append to a Physical Sequential Data Set
file	<i>>>oe</i>	Append to or Create an OpenExtensions Text File
file	<i>>>sfs</i>	Append to or Create an SFS File
file	<i>>>sfsslow</i>	Append to or Create an SFS File
file	<i>>mdsk</i>	Replace or Create a CMS File on a Mode
file	<i>>mvs</i>	Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set
file	<i>>oe</i>	Replace or Create an OpenExtensions Text File
file	<i>>sfs</i>	Replace or Create an SFS File
file	<i>aftfst</i>	Write Information about Open Files
file	<i>diskback</i>	Read a File Backwards
file	<i>diskfast</i>	Read, Create, or Append to a File
file	<i>diskrandom</i>	Random Access a File
file	<i>diskslow</i>	Read, Create, or Append to a File
file	<i>diskupdate</i>	Replace Records in a File
file	<i>filetoken</i>	Read or Write an SFS File That is Already Open
file	<i>ftp</i>	Connect to an FTP Server and Exchange Data
file	<i>hfs</i>	Read or Append File in the Hierarchical File System

Stage Selection Guide

file	<i>hfsreplace</i>	Replace the Contents of a File in the Hierarchical File System
file	<i>hfsstate</i>	Obtain Information about Files in the Hierarchical File System
file	<i>mdskback</i>	Read a CMS File from a Mode Backwards
file	<i>mdskfast</i>	Read, Create, or Append to a CMS File on a Mode
file	<i>mdskrandom</i>	Random Access a CMS File on a Mode
file	<i>mdskslow</i>	Read, Append to, or Create a CMS File on a Mode
file	<i>mdskupdate</i>	Replace Records in a File on a Mode
file	<i>members</i>	Extract Members from a Partitioned Data Set
file	<i>pdsdirect</i>	Write Directory Information from a CMS Simulated Partitioned Data Set
file	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
file	<i>sfsback</i>	Read an SFS File Backwards
file	<i>sfsrandom</i>	Random Access an SFS File
file	<i>sfsupdate</i>	Replace Records in an SFS File
file	<i>state</i>	Provide Information about CMS Files
file	<i>statew</i>	Provide Information about Writable CMS Files
file descriptor	<i>filedescriptor</i>	Read or Write an OpenExtensions File that Is Already Open
files	<i>getfiles</i>	Read Files
filter	<i>filterpack</i>	Manage Filter Packages
find	<i>asmfind</i>	Select Statements from an Assembler File as XEDIT Find
find	<i>asmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
find	<i>find</i>	Select Lines by XEDIT Find Logic
find	<i>nfind</i>	Select Lines by XEDIT NFind Logic
find	<i>strasmfind</i>	Select Statements from an Assembler File as XEDIT Find
find	<i>strasmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
find	<i>strfind</i>	Select Lines by XEDIT Find Logic
find	<i>strnfind</i>	Select Lines by XEDIT NFind Logic
fitting	<i>fitting</i>	Source or Sink for Copipe Data
format	<i>fmtfst</i>	Format a File Status Table (FST) Entry
format	<i>scm</i>	Align REXX Comments
FST	<i>fmtfst</i>	Format a File Status Table (FST) Entry
FST	<i>state</i>	Provide Information about CMS Files
FST	<i>statew</i>	Provide Information about Writable CMS Files
FTP	<i>ftp</i>	Connect to an FTP Server and Exchange Data
full pack	<i>diage4</i>	Submit Diagnose E4 Requests
full screen	<i>buildscr</i>	Build a 3270 Data Stream
full screen	<i>fullscr</i>	Full screen 3270 Write and Read to the Console or Dialled/Attached Screen
gateway	<i>overlay</i>	Overlay Data from Input Streams
gateway	<i>stem</i>	Retrieve or Set Variables in a REXX or CLIST Variable Pool
gateway	<i>var</i>	Retrieve or Set a Variable in a REXX or CLIST Variable Pool
gateway	<i>vardrop</i>	Drop Variables in a REXX Variable Pool
gateway	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
gateway	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
gather	<i>fanin</i>	Concatenate Streams
gather	<i>faninany</i>	Copy Records from Whichever Input Stream Has One
gather	<i>fanintwo</i>	Pass Records to Primary Output Stream
generate	<i>maclib</i>	Generate a Macro Library from Stacked Members in a COPY File
hash	<i>digest</i>	Compute a Message Digest
heartbeat	<i>beat</i>	Mark when Records Do not Arrive within Interval
help	<i>help</i>	Display Help for <i>CMS Pipelines</i> or DB2
hexadecimal	<i>xrange</i>	Write a Range of Characters
HFS	<i>filedescriptor</i>	Read or Write an OpenExtensions File that Is Already Open

hierarchical	<i><oe</i>	Read an OpenExtensions Text File
hierarchical	<i>>>oe</i>	Append to or Create an OpenExtensions Text File
hierarchical	<i>>oe</i>	Replace or Create an OpenExtensions Text File
hierarchical	<i>hfs</i>	Read or Append File in the Hierarchical File System
hierarchical	<i>hfsdirectory</i>	Read Contents of a Directory in a Hierarchical File System
hierarchical	<i>hfsquery</i>	Write Information Obtained from OpenExtensions into the Pipeline
hierarchical	<i>hfsreplace</i>	Replace the Contents of a File in the Hierarchical File System
hierarchical	<i>hfsstate</i>	Obtain Information about Files in the Hierarchical File System
hierarchical	<i>hfsxecute</i>	Issue OpenExtensions Requests
hold	<i>dam</i>	Pass Records Once Primed
host	<i>hostid</i>	Write TCP/IP Default IP Address
host	<i>hostname</i>	Write TCP/IP Host Name
host	<i>tso</i>	Issue TSO Commands, Write Response to Pipeline
http	<i>httpsplit</i>	Split HTTP Data Stream
HTTP	<i>urldeblock</i>	Process Universal Resource Locator
huge	<i>dfsrt</i>	Interface to DFSORT/CMS
hypertext	<i>httpsplit</i>	Split HTTP Data Stream
IEBCOPY	<i>iebcopy</i>	Process IEBCOPY Data Format
image	<i>combine</i>	Combine Data from a Run of Records
immediate	<i>immcmd</i>	Write the Argument String from Immediate Commands
IMSG	<i>emsg</i>	Issue Messages
information	<i>devinfo</i>	Write Device Information
information	<i>stfle</i>	Store Facilities List
information	<i>stsi</i>	Store System Information
inject	<i>literal</i>	Write the Argument String
inject	<i>strliteral</i>	Write the Argument String
insert	<i>insert</i>	Insert String in Records
insert	<i>sql</i>	Interface to SQL
interface	<i>ldrtbls</i>	Resolve a Name from the CMS Loader Tables
interface	<i>nucext</i>	Call a Nucleus Extension
interface	<i>rexx</i>	Run a REXX Program to Process Data
interface	<i>stem</i>	Retrieve or Set Variables in a REXX or CLIST Variable Pool
interface	<i>var</i>	Retrieve or Set a Variable in a REXX or CLIST Variable Pool
interface	<i>vardrop</i>	Drop Variables in a REXX Variable Pool
interface	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
interface	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
Internet	<i>tcpclient</i>	Connect to a TCP/IP Server and Exchange Data
Internet	<i>tcpdata</i>	Read from and Write to a TCP/IP Socket
Internet	<i>tcplisten</i>	Listen on a TCP Port
interrupt	<i>waitdev</i>	Wait for an Interrupt from a Device
interval	<i>delay</i>	Suspend Stream
inverse	<i>not</i>	Run Stage with Output Streams Inverted
IP	<i>hostbyaddr</i>	Resolve IP Address into Domain and Host Name
IP	<i>hostbyname</i>	Resolve a Domain Name into an IP Address
IP	<i>hostid</i>	Write TCP/IP Default IP Address
ISPF	<i>ispf</i>	Access ISPF Tables
ISPF	<i>listispf</i>	Read Directory of a Partitioned Data Set into the Pipeline
ISPF	<i>subcom</i>	Issue Commands to a Subcommand Environment
IUCV	<i>starmsg</i>	Write Lines from a CP System Service
IUCV	<i>starsys</i>	Write Lines from a Two-way CP System Service
join	<i>join</i>	Join Records

Stage Selection Guide

join	<i>joincont</i>	Join Continuation Lines
join	<i>snake</i>	Build Multicolumn Page Layout
key	<i>lookup</i>	Find Records in a Reference Using a Key Field
label	<i>frlabel</i>	Select Records from the First One with Leading String
label	<i>strfrlabel</i>	Select Records from the First One with Leading String
label	<i>strtolabel</i>	Select Records to the First One with Leading String
label	<i>strwhilelabel</i>	Select Run of Records with Leading String
label	<i>tolabel</i>	Select Records to the First One with Leading String
label	<i>whilelabel</i>	Select Run of Records with Leading String
labels	<i>asmfind</i>	Select Statements from an Assembler File as XEDIT Find
labels	<i>asmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
labels	<i>between</i>	Select Records Between Labels
labels	<i>find</i>	Select Lines by XEDIT Find Logic
labels	<i>inside</i>	Select Records between Labels
labels	<i>nfind</i>	Select Lines by XEDIT NFind Logic
labels	<i>notinside</i>	Select Records Not between Labels
labels	<i>outside</i>	Select Records Not between Labels
labels	<i>strasmfind</i>	Select Statements from an Assembler File as XEDIT Find
labels	<i>strasmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
labels	<i>strfind</i>	Select Lines by XEDIT Find Logic
labels	<i>strnfind</i>	Select Lines by XEDIT NFind Logic
level	<i>query</i>	Query <i>CMS Pipelines</i>
line end	<i>block</i>	Block to an External Format
line end	<i>deblock</i>	Deblock External Data Formats
line feed	<i>block</i>	Block to an External Format
line feed	<i>deblock</i>	Deblock External Data Formats
lines	<i>count</i>	Count Lines, Blank-delimited Words, and Bytes
link	<i>diag4</i>	Submit Diagnose E4 Requests
list	<i>warplist</i>	List Wormholes
listener	<i>vmclisten</i>	Listen for VMCF Requests
LISTFILE	<i>wildcard</i>	Select Records Matching a Pattern
listing	<i>printmc</i>	Print Lines
listing	<i>sfsdirectory</i>	List Files in an SFS Directory
listing	<i>uro</i>	Write Unit Record Output
literal	<i>literal</i>	Write the Argument String
literal	<i>strliteral</i>	Write the Argument String
literal	<i>xrange</i>	Write a Range of Characters
loader tables	<i>ldrtbls</i>	Resolve a Name from the CMS Loader Tables
locate	<i>all</i>	Select Lines Containing Strings (or Not)
locate	<i>locate</i>	Select Lines that Contain a String
locate	<i>nlocate</i>	Select Lines that Do Not Contain a String
lookup	<i>ldrtbls</i>	Resolve a Name from the CMS Loader Tables
lookup	<i>nuext</i>	Call a Nucleus Extension
machine	<i>asatome</i>	Convert ASA Carriage Control to CCW Operation Codes
MACLIB	<i>maclib</i>	Generate a Macro Library from Stacked Members in a COPY File
MACLIB	<i>members</i>	Extract Members from a Partitioned Data Set
map	<i>diskid</i>	Map CMS Reserved Minidisk
map	<i>mapmdisk</i>	Map Minidisks Into Data spaces
mark	<i>juxtapose</i>	Preface Record with Marker
members	<i>structure</i>	Manage Structure Definitions

merge	<i>collate</i>	Collate Streams
merge	<i>gather</i>	Copy Records From Input Streams
merge	<i>juxtapose</i>	Preface Record with Marker
merge	<i>merge</i>	Merge Streams
message	<i>digest</i>	Compute a Message Digest
message	<i>emsg</i>	Issue Messages
message	<i>mqsc</i>	Issue Commands to a WebSphere MQ Queue Manager
message	<i>starmsg</i>	Write Lines from a CP System Service
message	<i>xmsg</i>	Issue XEDIT Messages
messages	<i>hlasmerr</i>	Extract Assembler Error Messages from the SYSADATA File
messages	<i>runpipe</i>	Issue Pipelines, Intercepting Messages
MIME	<i>qpdecode</i>	Decode to Quoted-printable Format
MIME	<i>qpencode</i>	Encode to Quoted-printable Format
mime	<i>64decode</i>	Decode MIME Base-64 Format
mime	<i>64encode</i>	Encode to MIME Base-64 Format
minidisk	<i><mdsk</i>	Read a CMS File from a Mode
minidisk	<i>>>mdsk</i>	Append to or Create a CMS File on a Mode
minidisk	<i>>mdsk</i>	Replace or Create a CMS File on a Mode
minidisk	<i>diage4</i>	Submit Diagnose E4 Requests
minidisk	<i>diskfast</i>	Read, Create, or Append to a File
minidisk	<i>diskid</i>	Map CMS Reserved Minidisk
minidisk	<i>diskslow</i>	Read, Create, or Append to a File
minidisk	<i>mapmdisk</i>	Map Minidisks Into Data spaces
minidisk	<i>mdskfast</i>	Read, Create, or Append to a CMS File on a Mode
minidisk	<i>trackread</i>	Read Full Tracks from ECKD Device
minidisk	<i>trackwrite</i>	Write Full Tracks to ECKD Device
mixed	<i>casei</i>	Run Selection Stage in Case Insensitive Manner
mixed	<i>zone</i>	Run Selection Stage on Subset of Input Record
monitor	<i>starmon</i>	Write Records from the *MONITOR System Service
MONWRITE	<i>starmon</i>	Write Records from the *MONITOR System Service
MQ	<i>mqsc</i>	Issue Commands to a WebSphere MQ Queue Manager
multicolumn	<i>snake</i>	Build Multicolumn Page Layout
multiple	<i>duplicate</i>	Copy Records
multiple	<i>unique</i>	Discard or Retain Duplicate Lines
multistream	<i>>>mdsk</i>	Append to or Create a CMS File on a Mode
multistream	<i>>>sfs</i>	Append to or Create an SFS File
multistream	<i>>>sfsslow</i>	Append to or Create an SFS File
multistream	<i>>mdsk</i>	Replace or Create a CMS File on a Mode
multistream	<i>>sfs</i>	Replace or Create an SFS File
multistream	<i>abbrev</i>	Select Records that Contain an Abbreviation of a Word in the First Positions
multistream	<i>all</i>	Select Lines Containing Strings (or Not)
multistream	<i>asmfind</i>	Select Statements from an Assembler File as XEDIT Find
multistream	<i>asmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
multistream	<i>between</i>	Select Records Between Labels
multistream	<i>chop</i>	Truncate the Record
multistream	<i>combine</i>	Combine Data from a Run of Records
multistream	<i>deal</i>	Pass Input Records to Output Streams Round Robin
multistream	<i>diskfast</i>	Read, Create, or Append to a File
multistream	<i>diskslow</i>	Read, Create, or Append to a File
multistream	<i>drop</i>	Discard Records from the Beginning or the End of the File
multistream	<i>fanin</i>	Concatenate Streams
multistream	<i>faninany</i>	Copy Records from Whichever Input Stream Has One
multistream	<i>fanintwo</i>	Pass Records to Primary Output Stream
multistream	<i>fanout</i>	Copy Records from the Primary Input Stream to All Output Streams

Stage Selection Guide

multistream	<i>fanoutwo</i>	Copy Records from the Primary Input Stream to Both Output Streams
multistream	<i>fillup</i>	Pass Records To Output Streams
multistream	<i>find</i>	Select Lines by XEDIT Find Logic
multistream	<i>frlabel</i>	Select Records from the First One with Leading String
multistream	<i>gather</i>	Copy Records From Input Streams
multistream	<i>inside</i>	Select Records between Labels
multistream	<i>locate</i>	Select Lines that Contain a String
multistream	<i>lookup</i>	Find Records in a Reference Using a Key Field
multistream	<i>maclib</i>	Generate a Macro Library from Stacked Members in a COPY File
multistream	<i>mdskfast</i>	Read, Create, or Append to a CMS File on a Mode
multistream	<i>mdskslow</i>	Read, Append to, or Create a CMS File on a Mode
multistream	<i>merge</i>	Merge Streams
multistream	<i>nfind</i>	Select Lines by XEDIT NFind Logic
multistream	<i>nlocate</i>	Select Lines that Do Not Contain a String
multistream	<i>notininside</i>	Select Records Not between Labels
multistream	<i>outside</i>	Select Records Not between Labels
multistream	<i>overlay</i>	Overlay Data from Input Streams
multistream	<i>pack</i>	Pack Records as Done by XEDIT and COPYFILE
multistream	<i>pick</i>	Select Lines that Satisfy a Relation
multistream	<i>spec</i>	Rearrange Contents of Records
multistream	<i>sql</i>	Interface to SQL
multistream	<i>state</i>	Verify that Data Set Exists
multistream	<i>state</i>	Provide Information about CMS Files
multistream	<i>statew</i>	Provide Information about Writable CMS Files
multistream	<i>strasmfind</i>	Select Statements from an Assembler File as XEDIT Find
multistream	<i>strasmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
multistream	<i>strfind</i>	Select Lines by XEDIT Find Logic
multistream	<i>strfrlabel</i>	Select Records from the First One with Leading String
multistream	<i>strnfind</i>	Select Lines by XEDIT NFind Logic
multistream	<i>strtolabel</i>	Select Records to the First One with Leading String
multistream	<i>strwhilelabel</i>	Select Run of Records with Leading String
multistream	<i>synchronise</i>	Synchronise Records on Multiple Streams
multistream	<i>take</i>	Select Records from the Beginning or End of the File
multistream	<i>tolabel</i>	Select Records to the First One with Leading String
multistream	<i>unique</i>	Discard or Retain Duplicate Lines
multistream	<i>update</i>	Apply an Update File
multistream	<i>verify</i>	Verify that Record Contains only Specified Characters
multistream	<i>whilelabel</i>	Select Run of Records with Leading String
multistream	<i>wildcard</i>	Select Records Matching a Pattern
MVS	<i>listcat</i>	Obtain Data Set Names
MVS	<i>sysdsn</i>	Test whether Data Set Exists
name	<i>hostbyaddr</i>	Resolve IP Address into Domain and Host Name
name	<i>hostbyname</i>	Resolve a Domain Name into an IP Address
netdata	<i>block</i>	Block to an External Format
netdata	<i>deblock</i>	Deblock External Data Formats
not find	<i>asmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
not find	<i>nfind</i>	Select Lines by XEDIT NFind Logic
not find	<i>strasmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
not find	<i>strnfind</i>	Select Lines by XEDIT NFind Logic
notation	<i>polish</i>	Reverse Polish Expression Parser
nucleus	<i>nuext</i>	Call a Nucleus Extension
number	<i>random</i>	Generate Pseudorandom Numbers

OpenExtensions	<i><oe</i>	Read an OpenExtensions Text File
OpenExtensions	<i>>>oe</i>	Append to or Create an OpenExtensions Text File
OpenExtensions	<i>>oe</i>	Replace or Create an OpenExtensions Text File
OpenExtensions	<i>filedescriptor</i>	Read or Write an OpenExtensions File that Is Already Open
OpenExtensions	<i>hfs</i>	Read or Append File in the Hierarchical File System
OpenExtensions	<i>hfsdirectory</i>	Read Contents of a Directory in a Hierarchical File System
OpenExtensions	<i>hfsquery</i>	Write Information Obtained from OpenExtensions into the Pipeline
OpenExtensions	<i>hfsreplace</i>	Replace the Contents of a File in the Hierarchical File System
OpenExtensions	<i>hfsstate</i>	Obtain Information about Files in the Hierarchical File System
OpenExtensions	<i>hfsxecute</i>	Issue OpenExtensions Requests
or	<i>combine</i>	Combine Data from a Run of Records
OS	<i><mvs</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set
OS	<i>>>mvs</i>	Append to a Physical Sequential Data Set
OS	<i>>mvs</i>	Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set
OS	<i>block</i>	Block to an External Format
OS	<i>deblock</i>	Deblock External Data Formats
OS	<i>iebcopy</i>	Process IEBCOPY Data Format
OS	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
OUTDESC	<i>sysout</i>	Write System Output Data Set
output	<i>eofback</i>	Run an Output Device Driver and Propagate End-of-file Backwards
overlay	<i>combine</i>	Combine Data from a Run of Records
overlay	<i>overlay</i>	Overlay Data from Input Streams
overstrikes	<i>c14to38</i>	Combine Overstruck Characters to Single Code Point
overstrikes	<i>overstr</i>	Process Overstruck Lines
pack	<i>pack</i>	Pack Records as Done by XEDIT and COPYFILE
pack	<i>unpack</i>	Unpack a Packed File
package	<i>filterpack</i>	Manage Filter Packages
pad	<i>pad</i>	Expand Short Records
page	<i>snake</i>	Build Multicolumn Page Layout
parser	<i>polish</i>	Reverse Polish Expression Parser
partition	<i>frtarget</i>	Select Records from the First One Selected by Argument Stage
partition	<i>totarget</i>	Select Records to the First One Selected by Argument Stage
partitioned	<i><mvs</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set
partitioned	<i>iebcopy</i>	Process IEBCOPY Data Format
partitioned	<i>listispf</i>	Read Directory of a Partitioned Data Set into the Pipeline
partitioned	<i>listpds</i>	Read Directory of a Partitioned Data Set into the Pipeline
partitioned	<i>members</i>	Extract Members from a Partitioned Data Set
partitioned	<i>pdsdirect</i>	Write Directory Information from a CMS Simulated Partitioned Data Set
partitioned	<i>readpds</i>	Read Members from a Partitioned Data Set
partitioned	<i>writepds</i>	Store Members into a Partitioned Data Set
pattern	<i>wildcard</i>	Select Records Matching a Pattern
pause	<i>pause</i>	Signal a Pause Event
PDS	<i>pdsdirect</i>	Write Directory Information from a CMS Simulated Partitioned Data Set
pipeline	<i>pipcmd</i>	Issue Pipeline Commands
pipeline	<i>runpipe</i>	Issue Pipelines, Intercepting Messages
Polish	<i>polish</i>	Reverse Polish Expression Parser
POSIX	<i><oe</i>	Read an OpenExtensions Text File
POSIX	<i>>>oe</i>	Append to or Create an OpenExtensions Text File
POSIX	<i>>oe</i>	Replace or Create an OpenExtensions Text File
POSIX	<i>filedescriptor</i>	Read or Write an OpenExtensions File that Is Already Open
POSIX	<i>hfs</i>	Read or Append File in the Hierarchical File System
predicate	<i>not</i>	Run Stage with Output Streams Inverted
predicate	<i>predselect</i>	Control Destructive Test of Records

Stage Selection Guide

print	<i>sysout</i>	Write System Output Data Set
printer	<i>printmc</i>	Print Lines
printer	<i>uro</i>	Write Unit Record Output
printer	<i>xab</i>	Read or Write External Attribute Buffers
pseudorandom	<i>random</i>	Generate Pseudorandom Numbers
punch	<i>punch</i>	Punch Cards
punch	<i>sysout</i>	Write System Output Data Set
punch	<i>uro</i>	Write Unit Record Output
QSAM	<i><mvs</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set
QSAM	<i>>>mvs</i>	Append to a Physical Sequential Data Set
QSAM	<i>>mvs</i>	Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set
QSAM	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
query	<i>diage4</i>	Submit Diagnose E4 Requests
query	<i>fullscrq</i>	Write 3270 Device Characteristics
query	<i>fullscrs</i>	Format 3270 Device Characteristics
query	<i>query</i>	Query <i>CMS Pipelines</i>
queue	<i>mqsc</i>	Issue Commands to a WebSphere MQ Queue Manager
queue	<i>stack</i>	Read or Write the Program Stack
quoted-printable	<i>qpdecode</i>	Decode to Quoted-printable Format
quoted-printable	<i>qpencode</i>	Encode to Quoted-printable Format
random	<i>diskrandom</i>	Random Access a File
random	<i>diskupdate</i>	Replace Records in a File
random	<i>mdskrandom</i>	Random Access a CMS File on a Mode
random	<i>mdskupdate</i>	Replace Records in a File on a Mode
random	<i>random</i>	Generate Pseudorandom Numbers
random	<i>sfsrandom</i>	Random Access an SFS File
random	<i>sfsupdate</i>	Replace Records in an SFS File
range	<i>pick</i>	Select Lines that Satisfy a Relation
range	<i>substring</i>	Write substring of record
read	<i><</i>	Read a File
read	<i><mdsk</i>	Read a CMS File from a Mode
read	<i><sfs</i>	Read an SFS File
read	<i><sfsslow</i>	Read an SFS File
read	<i>diskback</i>	Read a File Backwards
read	<i>diskfast</i>	Read, Create, or Append to a File
read	<i>diskrandom</i>	Random Access a File
read	<i>diskslow</i>	Read, Create, or Append to a File
read	<i>fbaread</i>	Read Blocks from a Fixed Block Architecture Drive
read	<i>filetoken</i>	Read or Write an SFS File That is Already Open
read	<i>getfiles</i>	Read Files
read	<i>mdskback</i>	Read a CMS File from a Mode Backwards
read	<i>mdskfast</i>	Read, Create, or Append to a CMS File on a Mode
read	<i>mdskrandom</i>	Random Access a CMS File on a Mode
read	<i>mdskslow</i>	Read, Append to, or Create a CMS File on a Mode
read	<i>pdsdirect</i>	Write Directory Information from a CMS Simulated Partitioned Data Set
read	<i>sfsback</i>	Read an SFS File Backwards
read	<i>sfsrandom</i>	Random Access an SFS File
read	<i>trackread</i>	Read Full Tracks from ECKD Device
reader	<i>reader</i>	Read from a Virtual Card Reader
rearrange	<i>spec</i>	Rearrange Contents of Records
recode	<i>vchar</i>	Recode Characters to Different Length

record	<i>addrdw</i>	Prefix Record Descriptor Word to Records
record format	<i>asatmc</i>	Convert ASA Carriage Control to CCW Operation Codes
record format	<i>block</i>	Block to an External Format
record format	<i>deblock</i>	Deblock External Data Formats
record format	<i>fblock</i>	Block Data, Spanning Input Records
record format	<i>iebcopy</i>	Process IEBCOPY Data Format
record format	<i>mctoasa</i>	Convert CCW Operation Codes to ASA Carriage Control
record format	<i>pack</i>	Pack Records as Done by XEDIT and COPYFILE
record format	<i>unpack</i>	Unpack a Packed File
recursive	<i>sfsdirectory</i>	List Files in an SFS Directory
redundancy	<i>crc</i>	Compute Cyclic Redundancy Code
reformat	<i>parcel</i>	Parcel Input Stream Into Records
relational	<i>sql</i>	Interface to SQL
relational	<i>sqlselect</i>	Query a Database and Format Result
replace	<i>>mdsk</i>	Replace or Create a CMS File on a Mode
replace	<i>>sfs</i>	Replace or Create an SFS File
replace	<i>change</i>	Substitute Contents of Records
reserve	<i>diskid</i>	Map CMS Reserved Minidisk
resolution	<i>hostbyaddr</i>	Resolve IP Address into Domain and Host Name
resolution	<i>hostbyname</i>	Resolve a Domain Name into an IP Address
return	<i>aggrc</i>	Compute Aggregate Return Code
reverse	<i>polish</i>	Reverse Polish Expression Parser
reverse	<i>reverse</i>	Reverse Contents of Records
REXX	<i>rexx</i>	Run a REXX Program to Process Data
REXX	<i>rexxvars</i>	Retrieve Variables from a REXX or CLIST Variable Pool
REXX	<i>scm</i>	Align REXX Comments
REXX	<i>space</i>	Space Words Like REXX
REXX	<i>stem</i>	Retrieve or Set Variables in a REXX or CLIST Variable Pool
REXX	<i>var</i>	Retrieve or Set a Variable in a REXX or CLIST Variable Pool
REXX	<i>vardrop</i>	Drop Variables in a REXX Variable Pool
REXX	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
REXX	<i>varload</i>	Set Variables in a REXX or CLIST Variable Pool
REXX	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
row/column	<i>3277bfra</i>	Convert a 3270 Buffer Address Between Representations
SAM	<i><mvs</i>	Read a Physical Sequential Data Set or a Member of a Partitioned Data Set
SAM	<i>>>mvs</i>	Append to a Physical Sequential Data Set
SAM	<i>>mvs</i>	Rewrite a Physical Sequential Data Set or a Member of a Partitioned Data Set
SAM	<i>qsam</i>	Read or Write Physical Sequential Data Set through a DCB
SCBLOCK	<i>nuext</i>	Call a Nucleus Extension
search	<i>lookup</i>	Find Records in a Reference Using a Key Field
secure	<i>digest</i>	Compute a Message Digest
select	<i>abbrev</i>	Select Records that Contain an Abbreviation of a Word in the First Positions
select	<i>all</i>	Select Lines Containing Strings (or Not)
select	<i>asmfind</i>	Select Statements from an Assembler File as XEDIT Find
select	<i>asmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
select	<i>between</i>	Select Records Between Labels
select	<i>drop</i>	Discard Records from the Beginning or the End of the File
select	<i>find</i>	Select Lines by XEDIT Find Logic
select	<i>flabel</i>	Select Records from the First One with Leading String
select	<i>inside</i>	Select Records between Labels
select	<i>locate</i>	Select Lines that Contain a String
select	<i>nfind</i>	Select Lines by XEDIT NFind Logic
select	<i>nlocate</i>	Select Lines that Do Not Contain a String

Stage Selection Guide

select	<i>notinside</i>	Select Records Not between Labels
select	<i>outside</i>	Select Records Not between Labels
select	<i>pick</i>	Select Lines that Satisfy a Relation
select	<i>predselect</i>	Control Destructive Test of Records
select	<i>sql</i>	Interface to SQL
select	<i>sqlselect</i>	Query a Database and Format Result
select	<i>strasmfind</i>	Select Statements from an Assembler File as XEDIT Find
select	<i>strasmnfind</i>	Select Statements from an Assembler File as XEDIT NFind
select	<i>strfind</i>	Select Lines by XEDIT Find Logic
select	<i>strfrlabel</i>	Select Records from the First One with Leading String
select	<i>strnfind</i>	Select Lines by XEDIT NFind Logic
select	<i>strtolabel</i>	Select Records to the First One with Leading String
select	<i>strwhilelabel</i>	Select Run of Records with Leading String
select	<i>take</i>	Select Records from the Beginning or End of the File
select	<i>tolabel</i>	Select Records to the First One with Leading String
select	<i>unique</i>	Discard or Retain Duplicate Lines
select	<i>verify</i>	Verify that Record Contains only Specified Characters
select	<i>whilelabel</i>	Select Run of Records with Leading String
select	<i>wildcard</i>	Select Records Matching a Pattern
selection	<i>casei</i>	Run Selection Stage in Case Insensitive Manner
selection	<i>frtarget</i>	Select Records from the First One Selected by Argument Stage
selection	<i>totarget</i>	Select Records to the First One Selected by Argument Stage
selection	<i>zone</i>	Run Selection Stage on Subset of Input Record
server	<i>runpipe</i>	Issue Pipelines, Intercepting Messages
server	<i>vmclisten</i>	Listen for VMCF Requests
service	<i>filterpack</i>	Manage Filter Packages
service	<i>help</i>	Display Help for <i>CMS Pipelines</i> or DB2
service	<i>query</i>	Query <i>CMS Pipelines</i>
SFS	<i><sfs</i>	Read an SFS File
SFS	<i><sfsslow</i>	Read an SFS File
SFS	<i>>>sfs</i>	Append to or Create an SFS File
SFS	<i>>>sfsslow</i>	Append to or Create an SFS File
SFS	<i>>sfs</i>	Replace or Create an SFS File
SFS	<i>filetoken</i>	Read or Write an SFS File That is Already Open
SFS	<i>sfsback</i>	Read an SFS File Backwards
SFS	<i>sfsdirectory</i>	List Files in an SFS Directory
SFS	<i>sfsrandom</i>	Random Access an SFS File
SFS	<i>sfsupdate</i>	Replace Records in an SFS File
single	<i>unique</i>	Discard or Retain Duplicate Lines
SMART	<i>vmc</i>	Write VMCF Reply
sockaddr in	<i>ip2socka</i>	Build sockaddr_in Structure
sockaddr in	<i>socka2ip</i>	Format sockaddr_in Structure
socket	<i>ip2socka</i>	Build sockaddr_in Structure
socket	<i>socka2ip</i>	Format sockaddr_in Structure
sockets	<i>tcpclient</i>	Connect to a TCP/IP Server and Exchange Data
sockets	<i>tcpdata</i>	Read from and Write to a TCP/IP Socket
sockets	<i>tcplisten</i>	Listen on a TCP Port
sort	<i>dfsrt</i>	Interface to DFSORT/CMS
sort	<i>merge</i>	Merge Streams
sort	<i>sort</i>	Order Records
space	<i>space</i>	Space Words Like REXX
span	<i>block</i>	Block to an External Format
span	<i>deblock</i>	Deblock External Data Formats
span	<i>fblock</i>	Block Data, Spanning Input Records

spin	<i>sysout</i>	Write System Output Data Set
split	<i>spill</i>	Spill Long Lines at Word Boundaries
split	<i>split</i>	Split Records Relative to a Target
split	<i>threeway</i>	Split record three ways
SPOOL	<i>sysout</i>	Write System Output Data Set
SQL	<i>sql</i>	Interface to SQL
SQL	<i>sqlcodes</i>	Write the last 11 SQL Codes Received
SQL	<i>sqlselect</i>	Query a Database and Format Result
SSL	<i>ftp</i>	Connect to an FTP Server and Exchange Data
stack	<i>stack</i>	Read or Write the Program Stack
statements	<i>asmcont</i>	Join Multiline Assembler Statements
statements	<i>asmxpnd</i>	Expand Joined Assembler Statements
status	<i>jeremy</i>	Write Pipeline Status to the Pipeline
stop	<i>dam</i>	Pass Records Once Primed
stop	<i>gate</i>	Pass Records Until Stopped
stop	<i>pipestop</i>	Terminate Stages Waiting for an External Event
storage	<i>storage</i>	Read or Write Virtual Machine Storage
stream	<i>parcel</i>	Parcel Input Stream Into Records
streams	<i>synchronise</i>	Synchronise Records on Multiple Streams
string	<i>insert</i>	Insert String in Records
string	<i>xrange</i>	Write a Range of Characters
strip	<i>strip</i>	Remove Leading or Trailing Characters
structure	<i>structure</i>	Manage Structure Definitions
subcommand	<i>subcom</i>	Issue Commands to a Subcommand Environment
subset	<i>substring</i>	Write substring of record
substring	<i>substring</i>	Write substring of record
substring	<i>threeway</i>	Split record three ways
suspend	<i>delay</i>	Suspend Stream
SVC 202	<i>cms</i>	Issue CMS Commands, Write Response to Pipeline
symbolic	<i>structure</i>	Manage Structure Definitions
synchronise	<i>synchronise</i>	Synchronise Records on Multiple Streams
system	<i>stfle</i>	Store Facilities List
system	<i>tsi</i>	Store System Information
system	<i>sysvar</i>	Write System Variables to the Pipeline
tab	<i>retab</i>	Replace Runs of Blanks with Tabulate Characters
tab	<i>untab</i>	Replace Tabulate Characters with Blanks
table	<i>ispf</i>	Access ISPF Tables
table	<i>sql</i>	Interface to SQL
table	<i>sqlselect</i>	Query a Database and Format Result
tape	<i>tape</i>	Read or Write Tapes
TCP/IP	<i>hostbyaddr</i>	Resolve IP Address into Domain and Host Name
TCP/IP	<i>hostbyname</i>	Resolve a Domain Name into an IP Address
TCP/IP	<i>hostid</i>	Write TCP/IP Default IP Address
TCP/IP	<i>hostname</i>	Write TCP/IP Host Name
TCP/IP	<i>ip2socka</i>	Build sockaddr_in Structure
TCP/IP	<i>socka2ip</i>	Format sockaddr_in Structure
TCP/IP	<i>tcpcksum</i>	Compute One's complement Checksum of a Message
TCP/IP	<i>tcpclient</i>	Connect to a TCP/IP Server and Exchange Data
TCP/IP	<i>tcpdata</i>	Read from and Write to a TCP/IP Socket
TCP/IP	<i>tcplisten</i>	Listen on a TCP Port
TCP/IP	<i>udp</i>	Read and Write an UDP Port
terminal	<i>console</i>	Read or Write the Terminal in Line Mode
terminal	<i>fullscr</i>	Full screen 3270 Write and Read to the Console or Dialed/Attached Screen

Stage Selection Guide

test	<i>predselect</i>	Control Destructive Test of Records
TEXT	<i>apldecode</i>	Process Graphic Escape Sequences
TEXT	<i>aplencode</i>	Generate Graphic Escape Sequences
text unit	<i>block</i>	Block to an External Format
text unit	<i>deblock</i>	Deblock External Data Formats
tick	<i>juxtapose</i>	Preface Record with Marker
time	<i>delay</i>	Suspend Stream
time	<i>greg2sec</i>	Convert a Gregorian Timestamp to Second Since Epoch
time	<i>sec2greg</i>	Convert Seconds Since Epoch to Gregorian Timestamp
timeout	<i>beat</i>	Mark when Records Do not Arrive within Interval
timestamp	<i>timestamp</i>	Prefix the Date and Time to Records
tokenise	<i>tokenise</i>	Tokenise Records
trace	<i>trfread</i>	Read a Trace File
track	<i>ckddeblock</i>	Deblock Track Data Record
track	<i>trackblock</i>	Build Track Record
track	<i>trackdeblock</i>	Deblock Track
track	<i>trackread</i>	Read Full Tracks from ECKD Device
track	<i>tracksquish</i>	Squish Tracks
track	<i>trackverify</i>	Verify Track Format
track	<i>trackwrite</i>	Write Full Tracks to ECKD Device
track	<i>trackxpan</i>	Unsquish Tracks
transfer	<i>ftp</i>	Connect to an FTP Server and Exchange Data
translate	<i>xlite</i>	Transliterate Contents of Records
transport	<i>httpsplit</i>	Split HTTP Data Stream
transport	<i>vmcdata</i>	Receive, Reply, or Reject a Send or Send/receive Request
triple DES	<i>cipher</i>	Encrypt and Decrypt Using a Block Cipher
TRSOURCE	<i>trfread</i>	Read a Trace File
truncate	<i>chop</i>	Truncate the Record
TSO	<i>command</i>	Issue TSO Commands
TSO	<i>subcom</i>	Issue Commands to a Subcommand Environment
TSO	<i>sysvar</i>	Write System Variables to the Pipeline
TSO	<i>tso</i>	Issue TSO Commands, Write Response to Pipeline
TXTLIB	<i>members</i>	Extract Members from a Partitioned Data Set
type	<i>fullscrq</i>	Write 3270 Device Characteristics
UDP	<i>udp</i>	Read and Write an UDP Port
unicode	<i>utf</i>	Convert between UTF-8, UTF-16, and UTF-32
unique	<i>sort</i>	Order Records
unique	<i>unique</i>	Discard or Retain Duplicate Lines
unique	<i>unique</i>	Discard or Retain Duplicate Lines
unix	<i>greg2sec</i>	Convert a Gregorian Timestamp to Second Since Epoch
unix	<i>sec2greg</i>	Convert Seconds Since Epoch to Gregorian Timestamp
unpack	<i>unpack</i>	Unpack a Packed File
update	<i>diskupdate</i>	Replace Records in a File
update	<i>mdskupdate</i>	Replace Records in a File on a Mode
update	<i>sfsupdate</i>	Replace Records in an SFS File
update	<i>update</i>	Apply an Update File
url	<i>httpsplit</i>	Split HTTP Data Stream
URL	<i>urldeblock</i>	Process Universal Resource Locator
UTF-16	<i>utf</i>	Convert between UTF-8, UTF-16, and UTF-32
UTF-32	<i>utf</i>	Convert between UTF-8, UTF-16, and UTF-32
UTF-8	<i>utf</i>	Convert between UTF-8, UTF-16, and UTF-32

variable	<i>var</i>	Retrieve or Set a Variable in a REXX or CLIST Variable Pool
variable	<i>vardrop</i>	Drop Variables in a REXX Variable Pool
variable	<i>varfetch</i>	Fetch Variables in a REXX or CLIST Variable Pool
variable	<i>varset</i>	Set Variables in a REXX or CLIST Variable Pool
variables	<i>configure</i>	Set and Query <i>CMS Pipelines</i> Configuration Variables
variables	<i>rexxvars</i>	Retrieve Variables from a REXX or CLIST Variable Pool
variables	<i>sysvar</i>	Write System Variables to the Pipeline
variables	<i>varload</i>	Set Variables in a REXX or CLIST Variable Pool
verify	<i>verify</i>	Verify that Record Contains only Specified Characters
VMCF	<i>vmc</i>	Write VMCF Reply
VMCF	<i>vmcdata</i>	Receive, Reply, or Reject a Send or Send/receive Request
VMCF	<i>vmclient</i>	Send VMCF Requests
VMCF	<i>vmclisten</i>	Listen for VMCF Requests
volume	<i>diage4</i>	Submit Diagnose E4 Requests
wait	<i>dam</i>	Pass Records Once Primed
wait	<i>delay</i>	Suspend Stream
wait	<i>gate</i>	Pass Records Until Stopped
wait	<i>waitdev</i>	Wait for an Interrupt from a Device
warp	<i>warp</i>	Pipeline Wormhole
web	<i>httpsplit</i>	Split HTTP Data Stream
web	<i>urldeblock</i>	Process Universal Resource Locator
word	<i>space</i>	Space Words Like REXX
word	<i>spill</i>	Spill Long Lines at Word Boundaries
words	<i>count</i>	Count Lines, Blank-delimited Words, and Bytes
words	<i>split</i>	Split Records Relative to a Target
wormhole	<i>warp</i>	Pipeline Wormhole
wormhole	<i>warplist</i>	List Wormholes
write	<i>></i>	Replace or Create a File
write	<i>>></i>	Append to or Create a File
write	<i>>>mdsk</i>	Append to or Create a CMS File on a Mode
write	<i>>>sfs</i>	Append to or Create an SFS File
write	<i>>>sfsslow</i>	Append to or Create an SFS File
write	<i>>mdsk</i>	Replace or Create a CMS File on a Mode
write	<i>>sfs</i>	Replace or Create an SFS File
write	<i>diskfast</i>	Read, Create, or Append to a File
write	<i>diskslow</i>	Read, Create, or Append to a File
write	<i>diskupdate</i>	Replace Records in a File
write	<i>fbawrite</i>	Write Blocks to a Fixed Block Architecture Drive
write	<i>filetoken</i>	Read or Write an SFS File That is Already Open
write	<i>mdskfast</i>	Read, Create, or Append to a CMS File on a Mode
write	<i>mdskslow</i>	Read, Append to, or Create a CMS File on a Mode
write	<i>mdskupdate</i>	Replace Records in a File on a Mode
write	<i>sfsupdate</i>	Replace Records in an SFS File
write	<i>trackwrite</i>	Write Full Tracks to ECKD Device
WWW	<i>httpsplit</i>	Split HTTP Data Stream
XEDIT	<i>subcom</i>	Issue Commands to a Subcommand Environment
XEDIT	<i>xedit</i>	Read or Write a File in the XEDIT Ring
XEDIT	<i>xmsg</i>	Issue XEDIT Messages
12-bit	<i>3277bfra</i>	Convert a 3270 Buffer Address Between Representations
14-bit	<i>3277bfra</i>	Convert a 3270 Buffer Address Between Representations

Stage Selection Guide

1403	<i>c14to38</i>	Combine Overstruck Characters to Single Code Point
1403	<i>optcdj</i>	Generate Table Reference Character (TRC)
1403	<i>overstr</i>	Process Overstruck Lines
1403	<i>xpndhi</i>	Expand Highlighting to Space between Words
3270	<i>apldecode</i>	Process Graphic Escape Sequences
3270	<i>aplencode</i>	Generate Graphic Escape Sequences
3270	<i>browse</i>	Display Data on a 3270 Terminal
3270	<i>buildscr</i>	Build a 3270 Data Stream
3270	<i>fullscr</i>	Full screen 3270 Write and Read to the Console or Dialed/Attached Screen
3270	<i>fullscrj</i>	Format 3270 Device Characteristics
3277	<i>3277bfra</i>	Convert a 3270 Buffer Address Between Representations
3277	<i>3277enc</i>	Write the 3277 6-bit Encoding Vector
3800	<i>c14to38</i>	Combine Overstruck Characters to Single Code Point
3800	<i>optcdj</i>	Generate Table Reference Character (TRC)
6-bit	<i>3277enc</i>	Write the 3277 6-bit Encoding Vector
8c	<i>fullscrq</i>	Write 3270 Device Characteristics

Appendix B. Messages, Sorted by Text

To enable you to obtain help for a message that was issued without the message number, the messages are here listed sorted on the contents, truncated in the right margin. Text set in *italics* font is replaced with a value when the message is issued.

Issue the command “pipe help” (with no operands) to invoke help for the last message issued. Use a numeric operand to obtain help for the ten messages issued before the last one.

```

1478I   Stage hex
1477I   Vector hex
1476I   Header hex
1475I   Thread hex
 341I   . hex: hex *char*
1018I   ... hex: hex char
  39I   ... Data: "data"
1426I   ... Evaluating "string"
 412I   ... GPRn: hex
 411I   ... In procedure; offset offset in module
  3I   ... Issued from stage number of pipeline number
  4I   ... Issued from stage number of pipeline number name "name"
 356I   ... Message parameter string
 702I   ... Parameter: hex
1441I   ... Processed number structures and number members in next structure
  2I   ... Processing "command"
 355I   ... RDS: number DBSS: number; number rows done; string
  1I   ... Running "string"
 192I   ... Scan at position number; previous data "string"
 361I   ... SQL processing: string
 369I   ... SQL statement prepared: string
 413I   ... Store hex: hex
581E   >> cannot append to a member
1341I   data data
1342I   data data data
1343I   data data data data
1344I   data data data data data
1345I   data data data data data data
1346I   data data data data data data data
1347I   data data data data data data data data
1348I   data data data data data data data data data
1349I   data data data data data data data data data data
1340E   message
 728I   number description
1299W   number duplicate masters were discarded
 186I   PIPMOD MSGLEVEL number
 107E   PIPMOD nucleus extension dropped before PIPE command is complete
 727I   string
 155E   "attribute" is not three characters or hexadecimal
1391E   "char" is not valid in identifier string
1512E   "char" is not valid in subscript of identifier string
  34I   "entry point" called
  65E   "string" is not hexadecimal
 704E   A component of path is not a directory (path "string", reason code hex)

```

Message texts

1180E A directory in the path *string* does not exist or you are not authorised for it
410E ABEND *code* at *address*; PSW *hex*
609E ABEND *code* reason code *number*
1225E ABEND *hex* accessing the global data area
1338E ABEND *hex* reason *number* on LOAD of *entry point*
1296I ABEND in CMS command. Last *number* lines of output follow
1239I About to receive from socket
1237I Active process and thread IDs:*hex* (hexadecimal)
574E Address is odd
1527E Address space *word* is not available for user *word*
1529E Address space name *word* already exists
1528E Address space name *word* is not valid
1519E Address space name longer than 24: *word*
1532E Address space size is not valid: *number*
1253E Address *X'hex'* before section base
1536W ALET *hex* is neither valid nor revoked.
1521E ALET *hex* is not valid
1549E ALET and PGMLIST are incompatible
1023E All application slots in use
1412E Allocation would require more than two gigabytes
1350E Already connected to queue manager *string*
562E Alternate exec processor *name*; return code *number*
563W ANYOF assumed in front of *string*
1573E Argument *number* is required
1452E Argument is a string, not a single character: *string*
371E ARIRVSTC TEXT is not available; run SQLINIT
667E Arithmetic overflow
1503E Array size is greater than 2G
1533W ASIT *hex* is already permitted to user *word*
1534W ASIT *hex* is already permitted to VCIT *hex*
1523E ASIT *hex* is not valid
1321I Assembler requests *number* bytes output for record *number* on stream *number*
409E Assert failure *code* at *address*
1083E Assignment is not to a counter
556E Asterisk cannot end output column range
1329E Attempt to extract the square root of a negative number
1234I Attention exit disabled. Hit attention to terminate command
1432E Bad placement option *string*
1567E Beginning block *number* is greater than ending block *number*
1564E Beginning block number *number* larger than device capacity *number*
337E Binary data missing after *prefix*
1298E Binary number too large for counter (reason *number*)
1569E Block *number* after last writable block
1568E Block *number* before first writable block
575E Block padded with *hex*; it should be *X'00'*
152E Block size *number* too large; *number* is the maximum
69E Block size mismatch; *number* bytes read, but block descriptor word contains *number*
114E Block size missing
75E Block size not integral multiple of record length; remainder is *number*
115E Block size too small; *number* is minimum for this type
1442E Both ranges specify same length as other *string*
1288I Branch to zero probably from *hex*
1084E BREAK items are not allowed after EOF item

536E Buffer header destroyed: *hex*
1276E Buffer length is not valid (*hex* doublewords requested)
88E Buffer overflow
735E Callable Services are not available
234E Caller not REXX
614E Caller's current input stream is not connected
616E Caller's producer is not blocked waiting for output
615E Caller's producer is not connected to caller
402I Calling Syntax Exit
1507E Cannot access entire varying bounds array *word*
1185E Cannot convert absolute date format *word* to relative date format *word*
1168E Cannot convert relative date format *word* to absolute date format *word*
1596W Cannot erase original file renamed to *fileid*. Try SFS device driver instead
1167I Cannot load message repository *word*
1578E Cannot obtain lock for COMMAND stage (held by process *number* thread *number*)
1579W Cannot release lock for COMMAND stage: *hex*
611E Cannot set CONSOLE exit
79E CCW command code *X'hex'* is not valid
1470E CCW length *number* differs from record length *number* module *word*
766E Century incorrect in timestamp: *string*
1374E CESD IDs descending *number* follows *number*
179E Character "*char*" is not an ASA carriage control character
180E Character *X'hex'* is not a machine carriage control character
1487E Checksum field in column *number* is not within record length *number*
1353E Cipher functions are not available *hex*
1352E Cipher Message instruction not available
1191I Close flags *string*
1192I Close flags *string*; record length *number*, count *number*
1112I Closing socket (reason *number*)
18E CMS Pipelines incorrectly generated with *character*
86I CMS Pipelines, 5741-A07 *modlevel* (Version.Release/Mod) - Generated *April 29, 2020 at 2:50 p.m.*
560I CMS Pipelines, 5741-A07 level *hex*
324E CMSIUCV application not active in server
193E Colon missing in connector
71E Column number "*number*" must be positive
196E Column ranges must be in ascending order and not overlapping
1462E Comma expected; found *string*
1443E Comma list is available only with equal compares
1444E Comma list is not available with implied length
1497E Comma or right parenthesis expected; end of member found
1498E Comma or right parenthesis expected; found *char*
540E Command is longer than 132 (*number* characters)
62E Command length *number* too long for CP
537I Commit level *number*
105E Compiler overflow
104E Compiler stack overflow
707E Component in path name is too long: path "*string*"
1129E Component of host name too long: *string*
1354E Computed output column is not positive (it is *number*)
1433E Computed output length is negative (it is *number*)
1335W Concatenated data set(s) for DD=*DDNAME* ignored. Use QSAM instead
347E Condition code 3 on IUCV instruction
1165E Configuration variable *name* is not recognised

Message texts

- 592E Conflicting allocation for data set *DSNAME*
- 48E Conflicting value for keyword *keyword: character*
- 656E Connection to *word* severed with code *word*
- 101E Connector *connector* can be specified with ADDPIPE or CALLPIPE
- 99E Connector not at the beginning or the end of a pipeline
- 98E Connector not by itself
- 1217I Contents: *hex*
- 1373E Control record requests *number* bytes, but only *number* bytes are available
- 392E Conversion error in routine 2: *type*, record 3: *number* (reason code 1: *reason*); data: "*4: string*"
- 1488E Convert index *number* is not implemented
- 1273E Count area incomplete (*number* bytes available)
- 198E Count must be one when first string is null
- 1050E Counter contains more digits than picture: *string*
- 1409E Counter exponent out of range for hexadecimal: *number*
- 1561E Counter number *number* is not valid (valid: *number* to *number*)
- 1085E Counter number expected
- 1039E Counter overflow
- 1424E Counter underflow
- 1255E CP paging error on diagnose 210 device number *hex*
- 308E CP system service *name* not valid
- 650E CP system service *word* is in use by another program
- 1147E Creation time cannot be changed for an existing file
- 734E CSL Routine *name* is not loaded
- 688I CSW *hex*; last CCW *hex*; some data *hex*
- 1375E CTL or RLD record expected, but found *hex*
- 1377E CTL record found as record X'*hex*', but count is X'*hex*'
- 1193I Current input stream *number* has record available
- 370E Cursor has been closed
- 1256E Cylinder number *number* beyond disk capacity *number*
- 253E Data not a NETDATA control record
- 1048E Data not packed decimal: X'*hex*'
- 504E Data set *DSNAME* does not exist
- 505E Data set *DSNAME* is not partitioned
- 500E Data set *DSNAME* is partitioned
- 1380E Data set *string* is not a program library; member *word*
- 596E Data set name too long: *name*
- 1554E Data space ALET *hex* cannot be written
- 1547W Data space ALET *hex* contains unexpected lock *word*
- 1550E Data space ALET *hex* contains unexpected lock *word*
- 1546E Data space ALET *hex* is in use; lock is *word*
- 1555E Data space ALET *hex* is not accessible
- 1545E Data space ALET *hex* is not initialised properly; eye-catcher is *word*
- 1551E Data space ALET *hex* is not locked
- 1552E Data space is fetch protected in key *hex* (PSW key *hex*)
- 1553E Data space is write protected in key *hex* (PSW key *hex*)
- 1183E Date cannot be converted; input date *word* is not valid
- 1182E Date format *word* cannot be used as an input date format
- 375E DB2 already connected to subsystem *word*
- 374E DB2 connection using plan *word* already active
- 651E DCSS *word* is not loaded
- 652E DCSS name *word* does not match the DCSS name already established
- 506E DDNAME *name* is permanently concatenated

580I DDNAME allocated: *word*
 605E DDNAME longer than 8 characters: *word*
 58E Decimal number expected, but "*word*" was found
 1360E Degenerate Triple DES key
 400E Delay *word* is not acceptable
 60E Delimiter missing after string "*string*"
 280E Delimiter 16M or longer
 362E DESCRIBE followed by "*word*"; must be SELECT
 24W Descriptor list for program "*command*" is not doubleword aligned; it is ignored
 530E Destructive overlap
 159E Device *address* no longer exists
 1310I Device *hex* has unsolicited status pending
 1308I Device *hex* is busy or has interrupt pending
 1537E Device *hex* is not a reserved minidisk
 1538E Device *hex* is not attached
 83E Device *word* does not exist
 82E Device address *word* is not hexadecimal
 1267E Device number *hex* is read only
 1589E Dictionaries provided are *number*, but *number* is expected
 1590E Dictionary size is not a multiple of 4096 (X'*number*')
 1591E Dictionary size is not a power of 2 (X'*number*')
 1593E Dictionary size is too large (*number*)
 1592E Dictionary size is too small
 761E Different key fields not allowed with AUTOADD
 1541E Digit "*character*" is not hexadecimal in string *number*
 100E Direction "*word*" not input or output
 164E Direction "*word*" not valid or not supported
 1181E Directory control directory *string* is accessed read only
 691E Directory is missing: *word*
 148E Directory pointer *number* not compatible with file of size *number*
 748E Disk *mode* is full
 1544W Dispatcher called with address space control *hex*
 668E Divisor is zero
 785E DMSOPBLK is not supported
 1421E DO expected; *word* was found
 539E Do not connect unused *side* stream *stream*
 1196E Do not connect unused input stream *stream*
 1197E Do not connect unused output stream *stream*
 1580E Do not convert numeric type member as if it were a string
 758W Do not double up relational operators
 1422E DONE expected; *word* was found
 585I ECBs posted: *number*; hit attention again to stall the pipeline
 291E End of tape on *device*
 1077E ENDIF expected; *word* was found
 1565E Ending block number *number* larger than device capacity *number*
 1260E Ending cylinder (*number*) lower than the beginning one
 772E Ending period in destination *word*
 584I Enter PIPESTOP, PIPESTALL, or immediate pipeline command
 185E Entry point *name* is not executable
 27E Entry point *word* not found
 42E Entry point missing
 662E Environment already specified (*keyword* is met)
 1494E Equal sign expected; end of member found

Message texts

1495E Equal sign expected; found *char*
1015E ERRNO *number*: *chars*
1233E ERRNO *string* reason *string* in *string*
26E Error *number* obtaining storage
1282E Error *number* on HNDIO
1601E Error *number* parsing URL at "*string*"
237E Error code X'*hex*' (return code *number*) from EXECCOMM
1445I Error in call to *function*: *string*
636E Error in encoded pipeline specification; reason code *number*
1339E Error opening *string* for *string*
124E Error reading file: Length of record is *number* but file has logical record length *number*
129E Error reading file: Premature end of file
112E Excessive options "*string*"
745E Existing record length is not *number*
1599E Expansion failed after *number* bytes (reason *number*)
1467E Expect >; found *char*
1230E Expect "begin", found *string*
1231E Expect "end", found *string*
1337E Expect CSQN205I; received *string*
1146E Expect OF; found: *word*
1513E Expect period after subscript of identifier *string*; found *word*
1148E Expected parameter token "sysv"; found "*word*"
1435I Expecting *string*
665E Exponent is not valid: *word*
1427E Exponent out of range: *number*
1407E Exponent overflow (*number*)
679E Exponent too large: *number*
1324E Expression evaluated to the number "*hex*"
1323E Expression evaluated to the string "*string*"
206E Expression missing
1122E Expression result is a string: *string*
10E Extended format parameter list is required
1036E Field *ID* is already defined
1033E Field *ID* is not defined
1491E Field identifier specified, but no further operands are present
1492E Field identifier specified, but no valid range found: *word*
1037E Field identifiers cannot be defined in break items
284E Field or string longer than 16M
561E File *file* is no longer in storage
749E File *file* is on OS or DOS minidisk
746E File *file* is open with incompatible intent
743I File "*file*"
146E File "*fn ft fm*" does not exist
142E File "*fn ft fm*" is not in the XEDIT ring
740E File "*words*" does not exist or you are not authorised for it
700E File descriptor *number* is not open (reason code *hex*)
617E File does not have fixed format records; do not specify *keyword*
215E File identifier "*file*" not complete or too long
790E File locked by other user or other unit of work
126E File mode * not allowed
421E File mode *string* more than one character
117E File mode "*word*" longer than two characters
125E File mode missing
147E File not a proper PDS

121E File not found in the active file table
 701E File or directory does not exist (path "*string*" reason code *hex*)
 788E File pool is not available
 706E File system is quiescing (path "*string*")
 763E File token *word* is not valid (reason code *number*)
 116E File type missing
 791E File was committed by other user or other unit of work
 1315E Filter package *word* has bad eye-catcher *word*
 1312E Filter package *word* is already loaded
 1318E Filter package *word* is in use by *number* stages
 1316E Filter package *word* is not loaded
 1317E Filter package *word* is not loaded by FILTERPACK LOAD
 1319E Filter package cannot be loaded globally (task is not job step)
 1575E First argument to D2C/D2X is negative, but second argument is omitted: *number*
 220E First record not a delimiter: "*data*"
 1306E First record on track not 5 bytes long (it is *number*)
 723I Fitting *identifier* not resolved
 695E Fitting already defined: "*name*"
 792E Fitting placement incompatible with RPL
 1406E Fixed number needs at least *number* columns
 74E Fixed records not same length; *last* bytes followed by *current* bytes
 1436E FIXED specified, but no record length specified and no input
 1405E Floating point number too long (length *number*)
 1404E Floating point number too short (length *number*)
 778E Forbidden character in file name or file type *words*
 798I Forcing pipeline stall
 1368E Format character '*char*' not valid
 1413E Found *number* columns
 1414E Found *number* rows
 334E FROM value not valid for file of size *number* records
 1604I FTP *word* "*data*"
 1600E FTP error: *string*
 1603E FTP processing error *number*
 240E Function *name* not supported
 1078E Function does not support arguments; *word* was found
 1303E Function name expected, but identifier found: *number*
 711E Function not supported: *word*
 1079E Function requires one-character argument; "*word*" was found
 1474I Global *hex*
 1226E Global area is corrupted
 1358W Global lock held by R12=*address* R14=*address*
 1597E GLOBALV service not available
 1211I Got *hex* doublewords at *hex*
 298I HCPSGIOP contents: *hex*
 172E Help not available for relative message *number*; issue PIPE HELP MENU for the Pipelines help menu
 1040E Hex data too long (*number* bytes)
 64E Hexadecimal data missing after *prefix*
 1542E Hexadecimal string too long: *number*
 586I Hit attention again to terminate waiting stages
 643E HLASM not found in storage
 1134E Host *word* does not exist
 1135E Host *word* does not exist
 1535E Host access list is full

Message texts

1127E Host name too long: *string*
292E I/O error on *address*; CSW *X'hex'*, CCW *X'hex'*
1369E IDR record does not begin *X'80'*; found *X'char'*
1370E IDR record indicates *number* bytes present, but record is *number*
1222E IEANTRT RC=*hex*
1220E IEANTRT RC=*hex* not equal to R15 (=hex.)
1361E IEWBFDAT code *code* returns code *number* reason *X'hex'*
1322I Ignoring HALT at *hex*
717I Ignoring IUCV interrupt for message *number*; waiting for *number* (interrupt on path *number*; sent on *nu*)
587E Immediate command *name* is not active
23E Impossible record (*number* bytes from *X'address'*)
621W Impossible target string
1372E Improper control record prefix *hex*
1371E Improper IDR language processor flag byte *X'hex'* at offset *number*
1086E Improper operand for string expression
754E Improper use of stage; reason code *number*
1480I In procedure *word*
759E Incompatible types
81E Incomplete conversion triplet
1080E Incomplete IF
1396E Incomplete inputRange *string*
1390E Incomplete member definition: *name*
1460E Incomplete pattern
1511E Incomplete subscript in *string*
1251E Incomplete UTF-8 multibyte character
1423E Incomplete WHILE
221E Incorrect character "*character*" in expression
1175E Incorrect check word in PIPEBLOK: *word*
1250E Incorrect code point *X'hex'* in first byte
1252E Incorrect code point *X'hex'* in second byte
1598E Incorrect compression signature *X'hex'*
1254E Incorrect device number *hex* (larger than FFFF)
779E Incorrect directory *word*
582E Incorrect DSNAME "*string*"
742E Incorrect file "*file*" (reason code *number*)
777E Incorrect file mode number *word*
775E Incorrect file name *word*
781E Incorrect file token *hex*
776E Incorrect file type *word*
1387E Incorrect first character in identifier: *string*
1264E Incorrect home address *X'hex'*
750E Incorrect input block format
583E Incorrect member name "*string*"
1124E Incorrect NAMEDEF *word* (a directory name must contain a period)
68E Incorrect OS block descriptor word *X'hex'*
70E Incorrect OS record descriptor word *X'hex'*
768E Incorrect record in file; reading record *number*
1265E Incorrect record 0 count field *X'hex'*
1247E Incorrect selector value specified (interrupt code *number*)
1438I Incorrect text unit type *X'hex'*
1562E Incorrect UTF-*number* *X'hex'* reason code *number*
608E Incorrectly specified DSNAME *word*
1510E Index *number* is not positive
1506E Index *number* is out of bounds (*number*)

1504E Index missing for member *word*
 793E Initial RPL state is not valid: *number*
 1184E Input date *word* cannot be expressed in the output date format
 219E Input not in correct format (check word is "*check word*", not "*word*")
 1244E Input record contains incorrect data for ASCII quoted-printable format: X'*hex*'
 576E Input record is *number* bytes; disk block size is *number* bytes
 352E Input record is *number* bytes; it should be *number*
 1019E Input record is shorter than 24 bytes (it is *number*)
 681E Input record length (*number*) is over the maximum allowed (*number*)
 546E Input record length *number* is too short; 11 is minimum
 1261E Input record too long (it is *number*)
 401E Input record too short (*number* bytes)
 1556E Input record too short for complete VMCMHDR (*number* bytes available; 40 required)
 33I Input requested for *number* bytes
 1285I Input stream *number* is only stream connected
 1378E Installation validation routine rejected SVC 99
 1227E Insufficient data returned by DMSGETDI; got *number* expect *number*
 122E Insufficient free storage
 1548E Insufficient space in the data space for *number* bytes
 289E Intervention required on *device*
 1585I Invoking CMS command *word* with header *hex* at *hex* in process *hex* thread *hex*
 340I IPARML: *message* (R0=*number*)
 1022I IPARML: *message* (R0=*number*)
 1014E IPAUDIT *hex*
 343E IPAUDIT is not zero: *hex*
 313E IPRCODE *number* received on IUCV instruction
 312I IPUSER: *hex*
 304E ISPF is not active
 555I Issue PIPE AHELP PIPE or PIPE AHELP MENU
 1594I Issuing wait to operating system
 306E IUCV application *name* already active (HNDIUCV RC=4)
 344I IUCV External Interrupt *type*
 317E IUCV is not available to CMS
 1114I IUCV reply *number* bytes
 1351E Key length *number* is not valid
 1144E Key/ID field is not anchored at the extremities of the input record (*number* before; *number* after)
 1166E Keyword *name* is not recognised for configuration variable *name*
 109E Keyword *word* is not a valid blocking format
 187E Keyword *word* must be LIFO or FIFO
 664E Keyword is not supported when stage is first: *word*
 47E Label *label* is already declared
 46E Label *label* not declared
 44E Label *string* is not valid
 19W Label "*word*" truncated to eight characters
 176E Language "*word*" not found
 175E Language table not generated
 705E Last character is a slash (path "*string*")
 16E Last character is escape character
 1516E Last character of identifier is a period: *word*
 641I Last connected output stream severed by its consumer
 1088W Last operation is not assignment
 72E Last record not complete
 573E Last text unit or GDF order not complete

Message texts

771E Leading period in destination *word*
1439E Left hand operand is a string
380E Left parenthesis missing
1302E Leftmost word of 32-bit counter *number* is not zero (*hex*)
1229E Length code *char* is not valid
1408E Length of output member (*number*) is above maximum *number*
729I Letting dispatcher wait
657E Limit of connections to *word* is reached
690E Logical drive was not found: *word*
59E Logical record length *number* is not valid
1150E Lost race for SCBWKWRD
622E Mask and string are not the same length
1530E Maximum number of address spaces is exceeded
1531E Maximum size of address spaces is exceeded
502E Member *name* already selected by allocation
1428E Member *name* has no type
1429E Member *name* has unsupported type *char*
1398E Member *name* not defined in structure *name*
507E Member *name* not found
1364W Member *word* has no sections
1499E Member *word* is a manifest constant
1509E Member *word* is a scalar
1508E Member *word* is an array
1500E Member *word* is not a manifest constant (it is a data member)
150E Member *word* not found
1389E Member already defined: *name*
1399E Member name further qualified with *name*
595E Member name is not allowed for this function
1431E Member name longer than 16M (it is *number*)
597E Member name or generation too long in DSNAME *name*
607E Member name too long in DDNAME *name*
189I Messages issued: *list*
405E Minimal C program tries to extend DSA
96E Missing *PIPMOD* operand
1082E Missing colon
1266E Missing end of track marker
200E Missing ending parenthesis in expression
1397E Missing identifier in qualified name: *word*
163E Missing keyword INPUT or OUTPUT
1465E Missing number at end of pattern: *string*
51E Missing operand after inputRange(s)
1461E Missing pattern at *string*
57E Missing right parenthesis after inputRanges
281W Mixed case command verb "*word*"
214E Mode *fm* is not accessed or not CMS format
119E Mode *letter* not available or read only
1570E Mode *word* does not refer to an SFS directory
713E Mode is not valid: *word*
1320E Module *word* contains a type 1 filter package; run it as a CMS command to install
158E Modulo must be positive (it is *number*)
654E Monitor is currently running in exclusive mode; shared request rejected
653E Monitor is currently running in shared mode; exclusive request rejected
1283E More than *number* CCWs in input record

- 678E More than fifteen exponent digits in picture *picture*
- 1049E More than one decimal point in data: *string*
- 794E More than one RPL refers to stage
- 92E More than ten key fields
- 56E More than 10 inputRanges specified
- 80E More than 255 conversion triplets specified
- 1041E Multiplication overflow
- 1131E Name server on port *number* at *IPaddress* timed out
- 1133E Name server query in wrong format
- 1132E Name server response is truncated
- 753E NAMEDEF too long in *string*
- 300E Namelist does not end
- 1223E Nametoken field *name* contains *value*; expect *value*
- 1232E Native sockets are not available (reason code *number*)
- 1572E Needle cannot be empty
- 233E No active EXECCOMM environment found
- 1517E No active qualifier for *word*
- 1587E No compression dictionary provided
- 139E No connection available to redefine for *connector*
- 501E No data set is allocated for *DDNAME*
- 730E No data sets found matching *DSNAME*
- 760E No data will be available for input field
- 752E No default file pool defined
- 676E No digits selected in picture *picture*
- 692E No diskette in drive: *word*
- 13E No ending *right parenthesis* for global options
- 677E No exponent digits in picture *picture*
- 1518E No identifier found
- 55E No inputRange(s) in list
- 1013E No IUCV paths can be connected
- 1463E No matching specified
- 346E No message found (id *number*)
- 0E No message text for message *number*
- 1279E No messages in queue, but interrupt received.
- 1543E No minidisk pool has been defined
- 1171W No output date format specified; the default output date format is the same as the input date format
- 256I No pipeline specified on *pipe* command
- 301E No position for last variable
- 90E No reader file available
- 166E No real device attached for *device*
- 1410E No record read from stream *number*
- 1173I No RPL to restore
- 726I No RPLs changed state
- 1125E No space left in PDS directory
- 373E No SQL stub module or DB2 not present in system
- 173E No stage found to run
- 1386E No structure name found
- 1577E No structures defined in caller
- 1393E No structures defined in pipeline set
- 1402E No structures defined in thread
- 558E No symbol table available
- 773E Node *word* is not defined to JES

Message texts

1301E Not a built-in function: *word*
50E Not a character or hexadecimal representation: *word*
1038E Not a decimal number: "*word*"
515E Not a decimal range: *word*
716E Not a dotted decimal network address: *word*
1174E Not a hexadecimal address *word*
655E Not a named saved segment: *word*
516E Not a record number or a range of record numbers: *word*
1032E Not a valid field identifier: *word*
319E Not authorised to communicate with *service*
557E Not authorised to obtain CP load map
747E Not authorised to read *file*
338E Not binary data: *string*
1430E Not hexadecimal: X'*string*'
767E Not numeric character in timestamp: *string*
715E Not octal: *word*
123E Not same ADT
1311I Not squished track reason *hex*
1454E Not valid packed data *hex*
382E Nothing specified within parentheses
604E Null DDNAME
599E Null DSNAME *name*
43E Null label
606E Null member name in DDNAME *name*
598E Null member name or generation in DSNAME *name*
11E Null or blank parameter list found
12E Null pipeline
627E Null program read from stream
1416E Null record read from stream *number*
17E Null stage found
157E Null string found
231E Null variable name
287E Number *number* cannot be negative
66E Number *number* is outside the valid range
1496E Number expected; end of member found
1574E Number is not an integer: *number*
1258E Number of tracks *number* beyond remaining device capacity *number*
335E Odd number of characters in hex data: *string*
1464E Odd number of nibbles (*number*) in pattern: *string*
53E Odd number of translate pairs
1363E Odd string length *number*
149E Offset is not smaller than modulo
755E Offset not shorter than width
744I Open flags *words*
782E Open intent is incompatible with stage position (intent is *char*)
685E OpenExtensions is not available (reason code *number*)
686E OpenExtensions return code *number* reason code *hex* function: *word*
703I Opening "*hex*"
1186W Operand *string* is ignored for input date format *word*
245W Operand *word* ignored
283W Operand *word* ignored with *console*
111E Operand *word* is not valid
95E Operand *word* is not valid for *PIPMOD*
151E Operand "*string*" is not range of characters or a delimited string

- 154E Operating environment not supported by stage
- 1091E Operator expected; found *word*
- 635E Option *word* conflicts with option *word*
- 14E Option *word* not valid
- 1451E Option string is not valid for function: *character*
- 1450E Option string is null
- 345E Originator *name* severed path *number*
- 35I Output *number* bytes
- 183E Output buffer overflow; *number* required
- 499E Output descriptor *name* is not defined
- 498E Output descriptor *name* is not valid
- 508E Output descriptor too long: *word*
- 1455E Output field is *number* bytes, but packed number requires *number* bytes to avoid truncation
- 393E Output field too short to contain field length
- 63E Output specification *word* is not valid
- 61E Output specification missing
- 693I Packages sent: *number*; packages received: *number*
- 1449E Pad character is a string, not a single character: *string*
- 1326E Pad is not a single character (it is *string*)
- 1540E Page number too large: *number*
- 559E Paging error reading symbol table
- 194E Parenthesis not supported in connector
- 612I Parmlist: *hex*
- 1434E Parse error in state *number*, unexpected *string* at offset *number*: "*string*"
- 1021I Path *number* is connected for *application*
- 342I Path *number* is connected to *service*
- 712E Path name is missing from the input record
- 708E Path name is too long: path "*string*"
- 569E Path to *service* severed (path *number*)
- 1466E Pattern longer than 32767 bytes
- 770E Period missing in destination *word*
- 673E Picture has more than one V: *picture*
- 670E Picture longer than 255 characters: *picture*
- 1145W PIPE command was issued from XEDIT, which truncates at or before 255 characters (use Address Com
- 1381I Pipeline *word* committed to *number* worst return code *number*
- 195E Pipeline cannot contain only a connector
- 694E Pipeline is not called from a driving program
- 93E Pipeline not installed as a nucleus extension; use PIPE command
- 613E Pipeline specification is not issued with CALLPIPE
- 29E Pipelines stalled
- 339E PIPSDEL return code *number*
- 568I PL/I: *message*
- 378E Plan *word* is not authorised
- 661E Please ask nicely
- 407E PLISTART or CEESTART is not present
- 1459E Plus or minus expected; found *string*
- 288I Posting ECB at *address*
- 724I Posting fitting *identifier* with *hex*
- 170E Prefix or suffix type connector not allowed
- 624E Premature end of expression
- 1403E Premature end of expression; term expected
- 224E Premature end of primary input stream; sequence number *number* not found
- 1437E Previous member did not establish a position for *word*
- 350E Primary key longer than secondary

Message texts

1275I Processing cylinder *number* track *number* record *number*
1490I Processing item number *number*: *string*
1194I Producer on input stream *number* has record available
535E Program check *code*
797E Program check code '*hex*'x on TIO to communications device
1595E Prologue not recognised (*hex*)
1243T PSTV at *address* corrupt: *string*
181E PSW mask and key are X'*hex*', not X'FFE0' or X'03E0'
1313E PTF filter package *word* is already loaded
1113I Purging IUCV message
1401E Qualifier contains member: *name*
538I Query state of *side* stream *stream*
1139I Query summary state of streams
54E Range "*numbers*" not valid
197E Range shorter than first string
564W Range(s) should be before keyword; put more than one in parentheses
1336E Reason *number* on *string*: *string*
1235I Reason Code *hex* (*hex*) *number* (*decimal*)
1016I Reason: *chars*:
1110I Received *number* bytes
517E Record *number* not present in file
518E Record *number* truncated
238E Record count "*word*" not zero or positive
1126E Record descriptor indicates *number* bytes, but minimum is *number*
1100E Record descriptor is too small (it contains *number*)
741E Record format "*character*" is not supported
128E Record format not existing file format *letter*
241E Record format or logical record length is not valid
1274E Record incomplete (*number* bytes available; *number* bytes required)
134E Record is *number* bytes, but format F file record length is *number*
1277E Record length (*number*) is not 8+keylength+datalength (*number*)
1415E Record length *number* is not a multiple of four (stream *number*)
514E Record length *number* is over the maximum 32767
78E Record length *number* is too much
1359E Record length *number* not multiple of cipher block size *number*
680E Record length is zero
140E Record longer than specified length *bytes* bytes
547E Record number *number* is beyond end-of-file
1566E Record size (*number* blocks) does not agree with block count *number* in record
1270E Record zero missing
1212I Rel *hex* doublewords at *hex*
687E Relational operator expected; found *word*
41E Request "*code*" not valid on service call to *module*
1210I Request for *hex* doublewords unsuccessful (from *hex*)
1249E Requested SYSIB information not available
145I Requesting *function* on *fn ft fm*
113E Required operand missing
722I Resolved fitting *identifier*
1172I Restoring fitting name *word*
719I Resuming pipeline
31I Resuming stage; return code is *number*
120E Return code *error number* from parameter list *function fn ft fm*
77I Return code *number*

553E Return code *number* calling IRXSUBCM *function*
 1228I Return code *number* erasing work file
 303E Return code *number* from *function*
 699E Return code *number* from *function* (file: *word*)
 311E Return code *number* from CMSIUCV *function*
 91E Return code *number* from CONSOLE *type* macro
 297E Return code *number* from diagnose X'A8'
 732E Return code *number* from DMSCSL
 579E Return code *number* from DYNALOC; reason *hex*
 310E Return code *number* from HNDIUCV
 659E Return code *number* from LINEWRT macro
 162E Return code *number* from NUCEXT
 108E Return code *number* from operation *operation* on tape *tape*
 118E Return code *number* from renaming the file
 354E Return code *number* from SQL, detected in module *module*
 601E Return code *number* from STFSMODE
 577E Return code *number* from STIMERM
 731E Return code *number* from SVC 26
 600E Return code *number* from TGET
 543E Return code *number* from VMCF: *string*
 144E Return code *number* from XEDIT operation
 143E Return code *number* from XEDIT state
 503E Return code *number* obtaining data set control block
 1520E Return code *number* on ADRSPACE/ALSERV/MAPMDISK diagnose
 1330E Return code *number* on diagnose E0 subcode *hex*
 637E Return code *number* on IDENTIFY for *entry point*
 420E Return code *number* reading or writing block *number* on disk *mode*
 513E Return code *number* reading or writing XAB (parameters *hex*)
 89E Return code *number* reading the virtual reader
 376E Return code *number* reason *hex* from call to DSNALI
 591E Return code *number* reason code *hex* from BLDL
 594E Return code *number* reason code *hex* from STOW
 733E Return code *number* reason code *number* from *routine*
 762E Return code *number* reason code *number* from TSO
 549E Return code *number*, reason code *number*, R0 *hex* from IRXINIT
 1136E Return code from name server: *number*
 1584E Return code 70 renaming the file. Use >SFS instead
 1586I Return from CMS command *word*
 1012E Return/condition code *number* on IUCV declare buffer
 1011E Return/condition code *number* on IUCV QUERY
 718I Returning to application
 725I Returning to the pipeline dispatcher
 40E REXX program *name* not found
 1440E Right hand operand is a string
 1501E Right parenthesis expected after array bound; found *char*
 1505E Right parenthesis expected after index
 381E Right parenthesis missing
 1376E RLD record expected, but found CTL *hex*
 738E Router did not resolve entry point
 721I RPL *hex*
 1479I Running: *string*
 789E SAFE can be specified only for PRIVATE work unit
 1456E Scale not numeric: *string*
 1457E Scale out of bounds: *number* (-32768 to 32767 is valid range)

Message texts

- 639E Scaling allowed with packed data only
- 1560I Scanned member: *string*
- 1327E Scanner jammed in state *number* in start condition *number*
- 212E Screen size *number* less than 1920 or greater than 16384
- 191E Second character of connector not a period
- 211E Second target missing
- 222E Secondary stream not defined
- 1000E Secondary vector too short for *epname* or entry not present; install current CMS Pipelines
- 209E Segment length *number* not 2 or more
 - 73E Segmentation flags not compatible; previous is X'*previous*' and current is X'*current*'
 - 36I Select *side* stream *number*
- 1195I Selecting input stream *number*
- 1571E Self-defining is too long (*number* bits): *string*
- 1458E Semicolon expected; found *string*
- 1468E Semicolon, colon, or comma expected; found *char*
- 1563E Senary stream is incompatible with *word*.
 - 293I Sense *data*
- 1111I Sent *number* bytes
- 225E Sequence *number* not found
- 229E Sequence error in input stream from *previous* to *new*
- 223E Sequence error in output file: *previous* to *new*
- 226E Sequence field length *length* too long; 15 is maximum
- 227E Sequence field not present in record; *number* bytes read
- 314E Server *user ID* is not available
- 315E Server has not declared a buffer
- 318E Server machine has too many connections
- 1287E Server responds without SF4
 - 38I Setting dispatcher exit to X'*address*'
 - 548I SEVER function requested for *side*
- 1286E SF4 is not specified
- 593E Shared data set *DSNAME* cannot be allocated exclusive
- 138E Short circuit not from input to output in *connector*
- 1238I Shutting down for write
 - 552I SHVBLOCK: *hex*
- 1115I Socket call for *type*
- 1020E Socket operation cancelled (message is purged)
- 784E Space quota exceeded
- 131E Specified logical record length does not match existing logical record length *number*
- 786E Specified work unit does not exist
 - 177I Spent *number* milliseconds in *routine*
- 1332E SPOOL file *number* contains CP trace data
- 1333E SPOOL file *number* does not contain CP trace data
- 1331E SPOOL file *number* does not exist
- 1334E SPOOL file *number* is in use
- 511E Spool file identifier *SFID* rejected by CP
- 510E Spool ID *SFID* not found or incompatible with reader
- 1269E Spurious end of track marker
- 365E SQL has no information about *topic*
- 359E SQL object already exists
- 363E SQL RC -205: Column *name* not found in *creator.table*
- 358E SQL RC -805: Access module *name* not found; refer to help for SQL to generate access module
- 357E SQL RC -934: Unable to find module *module*; run SQLINIT
- 282E Stage cannot be used with ADDPIPE
- 1120E Stage cannot run in CMS subset

1121E Stage cannot run while DOS is ON
 1198I Stage is active
 1486I Stage is flagged to stop. Forcing exit from *word*
 1485I Stage is flagged to stop. I/O old PSW and IOPSW fields are not the same. Type B to continue
 1484I Stage is flagged to stop. It is not summarily stoppable
 1483I Stage is flagged to stop. PSW in wait/free storage management.
 1481I Stage is flagged to stop. PSW not in CMS Pipelines code
 30I Stage is in state *state*
 1482I Stage is in the dispatcher; likely to stop on the way out.
 1576E Stage is not running in a subroutine pipeline
 565W Stage is obsolete; use *name* instead
 20I Stage returned with return code *number*
 28I Starting stage with save area at X'*address*' on commit level *number*
 683I STAX return code *number*
 232E Stem or variable name is too long; length is *number* bytes
 1242I STOPECB *hex* called
 32I Storage *address length*
 1215E Storage at *address* allocated *hex* doublewords; releasing *hex*
 534E Storage at *address* is not addressable
 1213E Storage at *address* is not on allocated chain
 533E Storage at *address* is protected
 184E Storage at *address* not released; R12 *hex* R14 *hex*
 1214E Storage at *address*; check word at *address* is destroyed
 1216E Storage at *address*; check word is destroyed
 783E Storage group space limit exceeded
 532E Storage key *hex* not acceptable
 103E Stream *identifier* not defined
 102E Stream *number* not defined
 174E Stream "*identifier*" is already defined
 178E Stream "*identifier*" is not found
 133E Stream "*word*" already prefixed
 132E Stream "*word*" already replaced
 554E Stream identifier *string* must not be numeric
 165E Stream identifier *word* not valid
 45W Stream identifier "*name*" truncated to four characters
 169E Stream identifier missing
 37I Streamnum *side* stream number intersection *number*
 182W String "*string*" ignored in command
 1448E String contains a character that is not hexadecimal *char*: *string*
 1447E String contains blank not on byte boundary: "*string*"
 1446E String contains leading or trailing blank: "*string*"
 1417E String length cannot be negative: *number*
 336E String length not divisible by 8: *string*
 156E String missing
 1087E String operand not acceptable to operator
 1418E String position cannot be zero
 1385E Structure *name* is empty
 1559E Structure *word* is not built in
 1388E Structure already defined: *name*
 1384E Structure name expected; found *string*
 1392E Structure not defined: *name*
 1400E Structure not further qualified: *name*
 1394E Structure still in use: *name* (*number* users)
 1284E Subchannel for device number *hex* is busy

Message texts

- 257E Subcommand environment *word* not found
- 1514E Subscript "*word*" is not valid in identifier *string*
- 379E Subsystem *name* is not up
- 377E Subsystem *word* is not defined
- 1241I Suppressed CP/CMS command: *string*
- 638I SVC 99 parameter list *hex*
- 1588E Symbol translation and format-1 sibling descriptors are mutually exclusive
- 250E Syntax error in expression
- 666E Syntax error in expression; reason code *number*
- 774E Syntax error: *explanation*
- 769E SYSOUT Class *char* is not a letter
- 333E System service *name* is in use
- 360E Table *table* does not exist
- 305E Table *word* is not open
- 290E Tape *address* is write protected
- 279E Tape identifier *word* not valid
- 626E Target data missing for *keyword*
- 625E Target expression missing
- 720I Terminating pipeline
- 1382E Tertiary stream not defined
- 640I Text unit *type data*
- 1179W The alternate pointer to the Contents Vector table has been restored from the primary pointer
- 190E The character cannot begin a stage
- 1291I The field ADMSCWR in NUCON is incorrect; found *hex*; display of ABEND information may be in jeo
- 1170E The input date is not valid: *word* (reason code *number*)
- 67E The number is incompatible with "*option*"
- 1178W The pointer to the Contents Vector table has been restored from the alternate pointer
- 1176W The pointer to the Contents Vector table is destroyed (reason code *number*); investigate VM61261
- 137E The string of operands is too long
- 1177E The system does not support date format *word*
- 1221I The TSO Pipelines name/token is not established
- 1075E THEN expected; *word* was found
- 1328E There is no default for the type argument
- 1289E Third level interrupt exit is already set at *hex*
- 309E This machine has too many IUCV connections
- 127E This stage cannot be first in a pipeline
- 87E This stage must be the first stage of a pipeline
- 1300E Time zone offset *number* is not valid (86399 is max)
- 644E Timestamp *word* not valid; reason code *number*
- 765E Timestamp too long:*string*
- 764E Timestamp too short:*string*
- 94E Token *token* is not valid for *PIPMOD*
- 1419E Too few arguments in function call
- 242E Too few arguments; *number* is minimum
- 366E Too few input streams
- 736E Too few parameters in call to *name* (*number* found)
- 1493E Too few streams are defined; *number* are present, but *number* are required
- 1411E Too few streams are defined; *number* are present, but three streams are needed
- 1420E Too many arguments in function call
- 243E Too many arguments; *number* is maximum
- 658E Too many concurrent STIMERM requests
- 1089E Too many counters
- 204E Too many ending parentheses in expression
- 1074E Too many nested IFs

1149E Too many parameter tokens found (second is "*word*")
 737E Too many parameters in call to *name* (*number* found)
 1539E Too many ranges to save
 1268E Too many records on track (*number*)
 264E Too many streams
 302E Too many variable names specified (*number*); maximum is 254
 236E Too much data for variable *name*
 787E Too much ESM data (*number* bytes)
 1515E Top level structure "*word*" cannot be subscripted
 1297I Trace table at *hex*
 1307E Track capacity exceeded
 1257E Track number *number* beyond cylinder capacity *number*
 1278E Track number is not specified in input record *number*
 1583I Trap dropped into CP Read
 1582I Trap issued the CMS command "*string*"
 1453I Trap issued the CP command "*string*"
 1581I Trap requested
 1224I TSO Pipelines global area is at *hex*
 1128E Two consecutive periods in host name: *string*
 682I TXTunit list *hex*
 550E Unable to access variables
 542E Unable to communicate with *user ID*
 1017E Unable to connect to *server*
 307E Unable to connect to *service*
 1489E Unable to convert from negative to unsigned
 1355E Unable to convert to integer. *number* digits in fraction
 1357E Unable to convert to integer. Exponent too large. (*number*)
 1356E Unable to convert to integer: *number*
 1163E Unable to declare exit
 1272E Unable to find DMSEXI
 21E Unable to find EXECCOMM for REXX
 1162E Unable to find module *name*
 663E Unable to generate delimiter for *variable name*
 572E Unable to load *file* (EXECLOAD return code *number*)
 1161E Unable to load module *name* (return code *number*)
 1362E Unable to load module *word* (ABEND code *HEX* reason *number* cause *number*)
 1314E Unable to load module *word* (return code *number*)
 1473E Unable to obtain global lock; held by *hex*
 364E Unable to obtain help from SQL (return code *number*)
 261E Unable to open *DDNAME*
 1602E Unable to open FTP data connection
 603E Unable to read directory for member *name*
 1141E Unable to resolve *word* (RXSOCKET did not return a result)
 1142E Unable to resolve *word* (RXSOCKET error *string*)
 1140E Unable to resolve *word* (RXSOCKET is not available)
 1143E Unable to resolve *word* (RXSOCKET Version 2 is required)
 671E Unacceptable character *character* in picture *picture*
 674E Unacceptable drifting sign in picture *picture*
 1123E Unacceptable input record length *number*
 714E Unacceptable interval *word*
 672E Unacceptable picture *picture*; *word* is incorrect (reason code *number*)
 509E Unacceptable spool file identifier *SFID*
 675E Unacceptable zero suppress/protect in picture *picture*
 1309E Undefined return code *number* from Diagnose A8 on device *hex*

Message texts

1081E Unexpected character *char*
1469E Unexpected end of module *word*
1383E Unexpected EOF on primary input
320E Unexpected IUCV interrupt with IPTYPE *type* on path *number*
570E Unexpected IUCV interrupt with IPTYPE *type* on path *number*
1076E Unexpected keyword: *word*
1137E Unexpected response record type: *number*
1379E Unexpected return code X'*hex*' on SVC 99
1240I Unknown CP/CMS command: *string*
52E Unknown translate table "*word*"
1395E Unqualified member name: *name*
1090E Unrecognised operator *word*
1325E Unrecognised option *string*
1472E Unrecognised PIPMOD immediate command: *word*
623E Unrecognised relational operator *word*
1471E Unrecognised STOP parameter: *word*
620W Unsupported code page *number*
660E Unsupported code page *number*
391E Unsupported conversion *type*
602E Unsupported data set organisation *hex*
710E Unsupported file type *number* (file descriptor "*number*")
709E Unsupported file type *number* (path "*string*")
230E Unsupported format "*type*"
1281E Unsupported IUCV message format
406E Unsupported language code *number* for entry point
110E Unsupported record in IEBCOPY unloaded data
1138E Unsupported RESOLVEVIA: *word*
684E Unsupported system variable *word*
1558E Unsupported VMCF function code *number*
1425W Use parentheses when using the result of an assignment: *string*
566W Use secondary output instead of stack
367E Use SQL CONNECT TO to identify the subsystem (Reason *hex*)
756W Use the := assignment operator instead of =
757W Use the ¬ operator instead of !
1525E User *word* is not logged on
590E User data length is over 62 or odd (it is *number*)
348I UserData *data*
97E Userword for *pipe* nucleus extension is zero
1271W Using obsolete version of *word word*
1236I USS return code *number* reason *hex* function: *word*
49E Value for keyword "*keyword*" is not acceptable
15E Value missing for keyword "*keyword*"
1130E Variable *name* is not defined in file *string*
1164W Variable *name* is not valid: *contents*
1169E Variable *word* is not a token set by SCANRANGE (reason code *number*)
235E Variable name is not valid: *word*
1524E VCIT *hex* does not represent a user that is logged in
571E Virtual device *device number* is in use by another stage
512E Virtual device *device* not a spooled printer
85E Virtual device *word* is not a supported real type
84E Virtual device *word* is not a supported virtual type
1522E Virtual machine is not in XC mode
1526E Virtual machine may not share address spaces
1010E VMCF CVT not found

541E VMCF is in use by another stage
1557W VMCF message arrived, but no listener is active
545E VMCF message rejected by user *user ID*
544I VMCPARMS: *hex*
1259E Volume does not have label specified
76I Waiting on ECB at X'*address*': *hex*
1366E Warp *word* already registered
1367E Warp *word* no longer registered
1365E Warp *word* not registered
531E Word must be 8 characters; it is *number*
1502E Word-style not supported with an array
689E Workstation file is missing: *word*
141E XEDIT not active
780E You are not allowed to write to *file*
167E You cannot READ from the second reading station
1246E You need a more modern VM (interrupt code *number*)
1245E You need pipeline version *hex* for this stage
642E ZONE already specified
368E 10 SQL stages already active
796E 370 accommodation must be turned on (CP SET 370ACCOM ON)
161E 64K or more inbound data

Appendix C. Implementing CMS Commands as Stages in a Pipeline

For those knowing CMS commands, it may be interesting to see how the known primitives can be implemented with *CMS Pipelines*. This chapter may also be useful if you wish to “add an option” to a CMS command. Here you find how to write the original command as a pipeline specification; often, all you have to do is to “open it up” and add stages.

If the CMS command produces console output, another approach may be to run the command with *cms* and post process the output in the pipeline.

Though most CMS commands control the CMS environment, the ones listed below process data and are thus potential pipelines.

These commands cannot be formulated simply using the device drivers and filters of *CMS Pipelines*: COMPARE, DISK, EXECUPDT, LISTIO, MACLIB (one can be generated), TAPE (there is a device driver), TAPPDS, TXTLIB (members can be extracted).

EXECMAP, IDENTIFY, LISTFILE, MACLIB MAP, MODMAP, NUCXMAP, QUERY, and TXTLIB MAP can be issued via *cms* and *command* to obtain the output and process it in the pipeline.

Copyfile

The simplest usage of COPYFILE (to copy a file from one minidisk or directory to another one) is written as a cascade of two *disk* device drivers. Figure 408 shows how to copy a file from one mode letter to another one (making the output file variable record format).

Figure 408. Copying a Disk File

```
copyfile some file a = = b
pipe < some file a|> some file b
```

Most of the options on COPYFILE to change the file format or contents are implemented as separate stages, described below.

FROM *drop* <n-1>.
 FRLABEL *flabel*, but the argument is not restricted to eight bytes.
 TOLABEL *tolabel*, similarly not restricted.
 FOR *take*.
 SPECS *spec*.
 OVLY *overlay*.
 RECFM Use FIXED or VARIABLE option on the *disk* device driver.
 LRECL *pad* or *strip*.
 TRUNC *chop* or *strip*.
 OLDDATE No equivalent function is available.
 PACK *pack*, though with care: see the usage notes for *pack*.
 UNPACK *unpack*.
 FILL *pad* or *strip*.
 EBCDIC *xlate*. Archaic.
 UPCASE *xlate* UPPER.
 LOWCASE *xlate* LOWER.
 TRANS *xlate*.
 SINGLE *fanin* and multiple streams concatenate files.

Execio

Most of the functions of EXECIO can be done with device drivers and filters. *CMS Pipelines* supports more devices than does EXECIO, notably tape and 3270 full screen; unit record device addresses can be specified to use other than the standard devices. These are the device drivers corresponding to the EXECIO device options:

DISKR	<i>disk</i> and < read a CMS file. Use <i>diskslow</i> FROM to start at a particular line. Use <i>diskback</i> to read a CMS file backwards. Use <i>diskrandom</i> to read a CMS file randomly.
DISKW	<i>disk</i> , >, and >> write a CMS file. Use <i>diskslow</i> FROM to start at a particular line. Use <i>diskupdate</i> to replace records in a file.
CARD	<i>reader</i> reads card images. Select lines with X'41' in column 1 with <i>find</i> and discard the first column with <i>spec</i> . /* CARD REXX, subroutine to read cards */ 'callpipe reader find' '41'x' spec 2-* 1 *:'
CP	<i>cp</i> .
PUNCH	<i>punch</i> .
PRINT	<i>printmc</i> is equivalent to PRINT with the carriage control option (PRINT). Use <i>asatomc</i> to ensure the carriage control is converted to machine carriage control. /* PRINT REXX; subroutine to generate CC */ parse arg cc if cc='' then cc='09' /* Write space */ 'callpipe *: spec x'cc '1 1-* 2 asatomc printmc'
EMSG	<i>emsg</i> .
STEM	<i>stem</i> .
VAR	<i>var</i> .
LIFO	<i>stack</i> LIFO.
FIFO	<i>stack</i> FIFO.
SKIP	<i>hole</i> .
STRING	<i>literal</i> .

Other EXECIO options are used to edit lines. The equivalent *CMS Pipelines* filters are:

FIND	<i>find</i> selects records with a leading string. EXECIO also stacks a line with the number of the line selected. <i>diskrandom</i> NUMBER provides the record number in the first ten columns. For sequential read from the beginning of the file, use <i>spec</i> to put the record number into each record. Look for the string in columns 11 and onward.
LOCATE	<i>locate</i> to search for a string.
AVOID	<i>nlocate</i> to search for lines without a string.
ZONE	Use the column range option on <i>locate</i> and <i>nlocate</i> ; offset the argument to <i>find</i> .
MARGINS	<i>spec</i> .
STRIP	<i>strip</i> TRAILING.
CASE	Use <i>xlate</i> UPPER to upper case lines.

Movefile

Combine device drivers; no filters are needed.

CMS Commands Formulated as Pipelines

Netdata

Part of the function of this command is available in *block* NETDATA and *deblock* NETDATA. *deblock* TEXTUNIT deblocks the information in control records.

Print

Printing files with carriage control is done by *printmc*, possibly in conjunction with *asatomc*. Though more scaffolding is needed, the crux of the matter is:

```
'pipe <' file '| asatomc | printmc'  
'cp sp e close'
```

Punch

For NOHeader option:

```
'pipe <' file '|punch'  
'cp sp d close'
```

The header can easily be built from the output from *state*.

Readcard

Partly available using *reader*. Batched files and the :READ control card to name the file need a little more work.

Type

Immediately available as < connected to *console*. Use *spec* to do cols nn-nn and hex options. *members* is used instead of *disk* to read a member of a library.

Update

Single level update is available with *update*. Multilevel update is done as a cascade of such stages.

Appendix D. Running Multiple Versions of *CMS Pipelines* Concurrently

This appendix describes how *CMS Pipelines* initialises itself and how you can use two or more versions of *CMS Pipelines* concurrently if you perform the initialisation explicitly.

Basic Initialisation

These steps are performed when the PIPE command is issued for the first time in a CMS session (assuming, for the moment, that the code is loaded from disk):

1. The PIPE MODULE is brought into the CMS transient area. This is a small bootstrap module.
2. The bootstrap module looks for the nucleus extension PIPMOD, which contains the main pipeline module. Initially, this module is not loaded as a nucleus extension.
3. The bootstrap module loads DMSPipe MODULE as a system nucleus extension under the name PIPMOD. The PIPE MODULE bootstrap included with the Runtime Library loads the PIPELINE MODULE instead.
4. The bootstrap issues PIPMOD INSTALL to make the main pipeline module initialise itself.
5. The main pipeline module declares a (user) nucleus extension for PIPE. This nucleus extension will process future PIPE commands.
6. The bootstrap module regains control when the main module has been initialised. The bootstrap clears out the name of the module loaded in the transient area (to avoid a recursion) and then reissues the original PIPE command to process the pipeline specification. This time the command is processed by the main pipeline module.

A CMS ABEND will cause the PIPE nucleus extension to be dropped, because it is a user extension; but the PIPMOD nucleus extension will remain installed, because it is a system extension. A subsequent PIPE command will then bypass step 3.

Initialisation of a Shared Segment

Because the main module is several hundred kilobytes in size, it is recommended that it be installed in a shared segment. Define the shared segment and create or modify the definition file to list the logical segments in the physical segment. Create the definition file for the logical segment.

This statement should be used to include the main pipeline module in a logical segment:

```
MODULE PIPELINE ( SYSTEM SERVICE IMMCMD NAME PIPMOD
```

The segment is generated with the SEGGEN command.

When the main pipeline module is installed in a shared segment, the segment is attached to the virtual machine by the CMS command SEGMENT LOAD. This is normally performed in the system profile before any pipeline specifications have been issued. The main module is now installed as a nucleus extension by CMS. Thus, step 3 is bypassed when the pipeline is initialised on the first PIPE command.

Coexistence

The Runtime Library version of *CMS Pipelines* can coexist with the version shipped as part of z/VM. The version used is determined by which of the two PIPE bootstrap modules is loaded into the transient area in step 1. Make sure the module you wish to use is first in the search order or installed as a nucleus extension.

You can perform part or all of this initialisation procedure by hand to install a different module or to use a command name other than PIPE.

Assuming you are running with the Program Offering level of *CMS Pipelines* and you wish to try some commands against the pipeline shipped in VM/ESA, you can issue these commands to make an EPIPE command:

```
nucxload epipmod dmsspipe (system service immcmd
epipmod install epipe
```

To use QPIPE to issue pipeline specifications to the Program Offering:

```
nucxload qpipmod pipeline (system service immcmd
qpipmod install qpipe
```

! Filter Packages

! To use a filter package with *CMS Pipelines*, the filter package must be associated with the
! pipeline module. When multiple pipeline modules are installed, the filter package must
! associated with the right pipeline module (or both).

: *CMS Pipelines* supports two types of filter packages:

- : • Type 1, which is available with CMS only. Such a package includes the FPLNXF glue
: module. It is installed by invoking it as a CMS command. The balance of this section
: discusses type 1 filter packages.
- : • Type 2 filter packages include the FPLNXG glue module. They are managed by
! *filterpack*. The filter package is associated with the pipeline module in which the
! *filterpack* stage runs.

: The glue code in FPLNXF assumes that the main pipeline module is installed as the nucleus
! extension PIPMOD. Explicit installation of a filter package is required to use it with a
! pipeline that resides in a different nucleus extension. Perform these steps to install the
! module FPACK as a filter package with a main pipeline module that is installed as EPIPMOD.
! As case is important in these commands, they should be issued from an EXEC that has
! issued the instruction Address Command.

1. Install the filter package module as a nucleus extension with SYSTEM and SERVICE attributes:

```
NUCXLOAD FPACK ( SYSTEM SERVICE
```

2. Invoke the filter package with the argument as shown:

```
FPACK 5785-RAC load EPIPMOD
```

3. The association from the main module must be broken before the nucleus extension that contains the filter package can be dropped (unless the main module itself is dropped before the filter package is):

```
FPACK 5785-RAC drop EPIPMOD
```

4. You can now drop the filter package:

NUCXDROP FPACK

! When the filter package must be associated with multiple pipeline modules, step 2 must be
! done for each pipeline module. Before unloading the filter package nucleus extension, step
! 3 must also be done for each pipeline module.

Warning: Results are unpredictable (but are likely to be catastrophic) if a filter package is dropped without the main pipeline module being notified. This leaves a dangling pointer to what was once the entry point table in the filter package. A program check is likely next time an entry point is to be resolved by that particular pipeline module.

Appendix E. Generating and Using Filter Packages with *CMS Pipelines*

This chapter introduces the concept of a *filter package*; it explains the original PRPQ way to generate filter packages; and it shows how to use filter packages.

For compatibility with earlier releases, z/VM supplies the two commands PIPGFTXT and PIPGFMOD to enable CMS users to generate filter packages.

The programming interfaces in a filter package are described in *CMS Pipelines: PIPE Command Programming Interface*, in particular for user written functions for *spec*.

Note for MVS Users

CMS file terminology is used in this chapter. On z/OS, the file type should be read as the file (DDNAME) allocated to a partitioned data set (PDS) and the file name should be read as the member name in this PDS. Sequential data sets are not supported by the utility programs that build filter packages.

Introduction

A filter package is a module that contains additional built-in programs. These programs can be written in Assembler or REXX; REXX programs can be compiled. Programs in filter packages must be reentrant.

Once the main pipeline module “knows” about a filter package, it can resolve programs to run in a pipeline from the programs that are contained in the filter package as well as from the built-in ones. The filter package contains an entry point vector which the main pipeline module uses to resolve programs in the filter package.

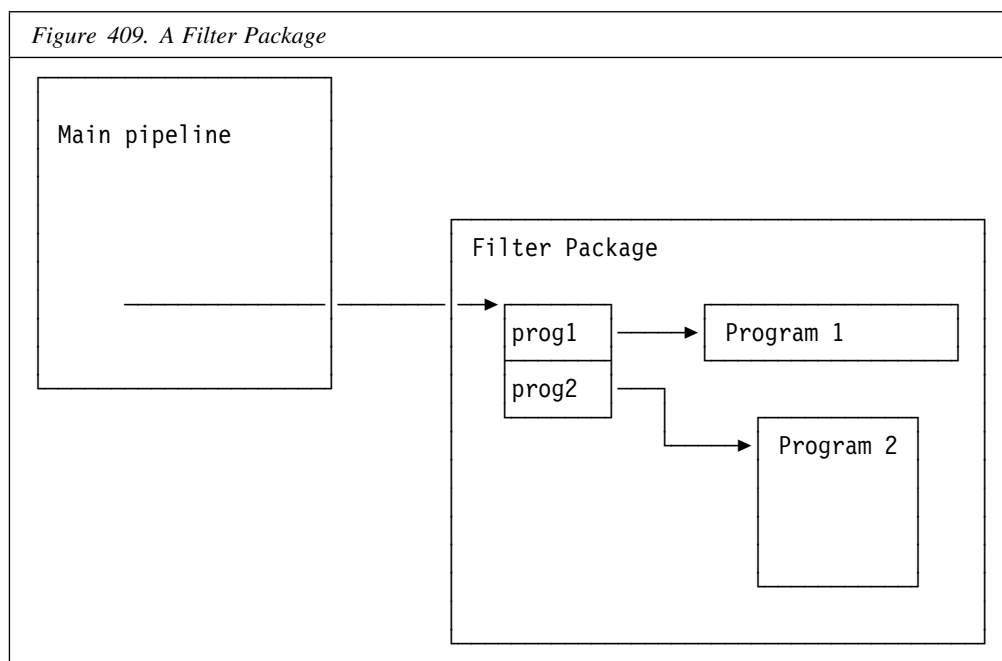
A filter package contains mostly code you supply, but the interface to *CMS Pipelines* is embodied in a bootstrap module.

Filter packages are available on CMS in two flavours, type 1 and type 2; and on z/OS in one flavour, type 2. The type 1 filter package is the original, which installs itself actively through the PIPMOD command. Once loaded, a type 1 filter package remains available until it is deleted explicitly. In contrast, a type 2 filter package is passive; it is loaded and deleted by the *fltpack* service program, though, on CMS, a second bootstrap module can be added to the filter package to make it self-installing and self-removing.

Filter packages are loaded as global or local to the thread (task). A global filter package is available to all pipelines within the virtual machine or address space, but a local filter package is available to pipelines on the thread (task) where it is loaded. On CMS, filter packages are by default loaded globally, whereas on z/OS, they are by default local unless the task is the job step task.

One filter package, the PTF *filter package*, receives special attention. Only one filter package can be the PTF filter package at any time.

The entry point vector is declared to the main pipeline module when the filter package’s main entry point is invoked as a CMS command (type 1) or when the filter package is explicitly loaded (type 2).



On CMS, four type 1 filter packages are installed in storage automatically when the main pipeline module initialises (and whenever you issue the command PIPMOD INSTALL). *CMS Pipelines* loads the modules as system nucleus extensions and attaches their entry point tables to its own built-in entry point table. The filter package modules that are installed automatically are those named PIPPTFF, PIPSYSF, PIPLOCF, and PIPUSERF. If PIPPTFF is available, it will be loaded as the PTF filter package. The entry point table in the PTF filter package is searched before the main module's entry point table; thus, filters in this package effectively override the built-in ones. All other filter packages are searched after the main module.

Additional filter packages must be installed manually. On CMS, a type 1 filter package is installed by issuing the file name of the filter package as a CMS command. This will make it install itself as a nucleus extension (if it is not already one) and declare itself to the main pipeline module.

Filter packages are managed by the *fltpack* control. This stage loads and deletes filter packages and also list installed filter packages of both types.

A type 1 filter package is detached from the pipeline when its nucleus extension is dropped, when the main pipeline module is dropped, and at CMS ABEND cleanup (HX).

z/OS contents management deletes all modules loaded by a task when the task terminates. This has the effect of deleting all filter packages loaded by the task.

Specifying Files

The utilities to generate filter packages use a notation for input and output files that specify the file name, type, and mode as a single word where the components are separated by periods. The file mode can specify an SFS directory on CMS.

Multiple input files are specified by concatenating the specifications for the individual files with a forward slash.

Filter Packages



When a default is applicable, its three components are applied independently. Specifically, when a default file mode is specified, it can be overridden only by an explicit file mode.

Contents of a Filter Package

In general, a filter package is made up from glue code supplied with *CMS Pipelines*, parameter files that you supply, and the actual filter programs, which you also supply. The individual components (which are separate object modules) are described in separate sections below.

The minimum filter package contains a glue code module, an entry point table, and the actual program to run. In addition, the filter package may contain a message text table and a keyword table. How to generate the object modules that contain your parameters (the entry point, keyword, and message text tables) is described below.

Once the object files have been generated, the filter package is generated on CMS by the CMS LOAD, INCLUDE, and GENMOD commands; on z/OS, the executable filter package module is created by the binder (linkage editor).

Except for the glue code module, the contents of a type 1 and a type 2 filter package is the same.

Glue Code

The object module PIPNXF TEXT contains the code that is invoked when a type 1 filter package module is invoked as a CMS command. This code is also invoked on service calls.

The type 2 glue module is FPLNXG TEXT. It may be combined with a third module, FPLNXH TEXT, which contains code to load the type-2 filter package so that it is operationally (but not as far as its contents are concerned) mimics a type-1 filter package.

All three modules are supplied in FPLLIB TXTLIB.

Entry Point Table

The entry point table is used by the pipeline specification parser to resolve programs within the filter package. The entry point table is generated from one or more source files that have the file type EPTABLE.

A source entry point table contains a line for each entry point and optionally comments. Comments begin with an asterisk and extend to the end of the line. Blank lines and lines

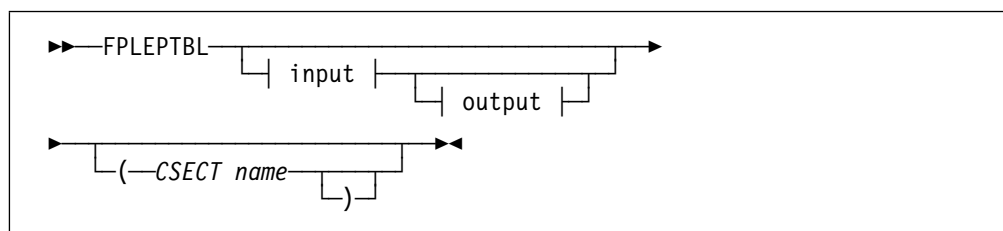
that contain only comments are ignored. Case is ignored in the source entry point table. Entry points are defined by blank-delimited words:

1. The name of the filter, as it would be used in a PIPE command.
2. The ESD name of the entry point for the program. This defaults to the name in the first word when the line contains one word.
3. A number specifying the minimum truncation that should be accepted for the filter. If the first word contains more than eight characters, you must specify a minimum truncation count and it must be a number that is 8 or less. 0 (the default) specifies that no truncation should be accepted.
4. The programming language in which the program is written or a period as a placeholder. The default is Assembler or REXX as determined by inspecting the program.
5. The commit level at which the program should start. The commit level can be specified as a number between -128 and 127 (inclusive). Be sure you know what you are doing if you specify this number positive, as this precludes using the program with the *CMS Pipelines* built-in programs.

A sample line of an entry point table:

```
console pipconep 4
```

FPLEPTBL—Generate Entry Point Table Object Module



The FPLEPTBL command generates an object entry point table from one or more source entry point tables. The FPLEPTBL command supports two blank-delimited words and an option:

1. Input file specifications. The default input file is SYSTEM EPTABLE on CMS; it is FPLEPT FPLPARMS, the member FPLEPT of the data set allocated to FPLPARMS.
2. The output file (which contains a single object module). By default, the output file is FPLEPT TEXT.
3. The ESD name of the control section that will contain the object entry point table is specified after a left parenthesis. The default is the file name of the output file or its default. For a type 1 filter package, it must be specified as PIPEPT; for a type 2 filter package, it must be specified as FPLEPT.

Message Text Table

The message text table contains the message texts for messages that are issued by the PIPERM macro and the ISSUMSG pipeline command. When *CMS Pipelines* resolves a message, it looks in its internal message text table and in the message text tables of all attached filter packages in this order:

1. The message text table in the PTF filter package, if one is installed and it contains a message text table.

Filter Packages

2. The message text table in the filter package where the stage that issues the message was resolved, if any.
3. The main message text table in module FPLMTX, which is linked with the PIPELINE module.
4. The message text table in each attached filter package. The packages are searched in the order they were attached. Thus, it would be normal to search PIPSYSF before PIPLOCF and PIPUSERF.

The source message text table is in BookMaster format. It often contains the information required to build a manual or help file, or both, in addition to the message text tags. FPLMSGTB processes the tags `:msgno` and `:msg`. All other lines and tags are ignored. For each message, the tags must be specified in the order `:msgno` followed by `:msg`. The two tags may be on the same line or separate lines, but no other tag may follow them on the same line. That is, the parsing of GML is simplistic. The `:msgno` tag specifies the message number and one character for severity code. If the severity code is lower case or the number 0 (zero), no additional identification messages are issued; if it is the number zero the message is not entered into the message list. The message text and substitution items are specified with the `:msg` tag.

Use DCF variables for characters that would interfere with the markup:

Semicolon `&semi`.

Colon `&colon`.

Ampersand `&`.

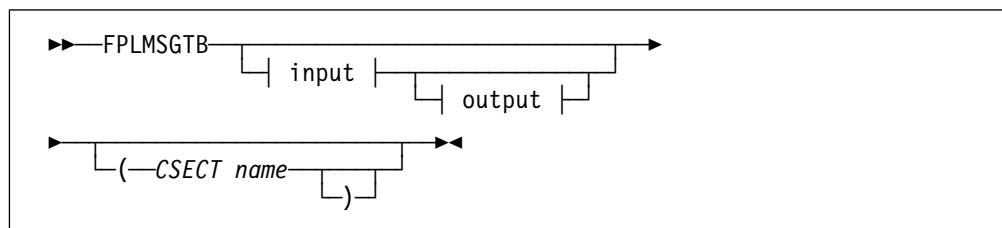
Quote `&csq`. However, this will be converted to a normal single quote (X'7D').

Within the message text, substitution is indicated by text that is bracketed within `:mv` and `:emv` tags. No other tags are allowed in the message text. By default, the items are substituted in the order they occur. A message value (a substitution item) is associated with a particular substitution if a number follows the opening tag; a colon must follow the number.

A sample entry in a message text table:

```
:msgno.900E
:msg.This is the first test message. (:mv.word:emv.)
```

FPLMSGTB—Generate Message Text Table Object Module



The FPLMSGTB command generates an object message text table from one or more source message text tables. It supports two blank-delimited words and an option:

1. Input file specification. The default is FPLMSG SCRIPT.
2. The output file, which will contain a single object module. The default output file is FPLMTX TEXT.

- The ESD name of the control section that will contain the object message text table may be specified after a left parenthesis. The default is the file name of the output file or its default. For a type 1 filter package, this must be specified as PIPMTX; for a type 2 filter package it must be specified as FPLMTX.

Keyword Table

The keyword table contains keyword definitions that are tested by programs in the filter package. REXX filters cannot access entries in the keyword table.

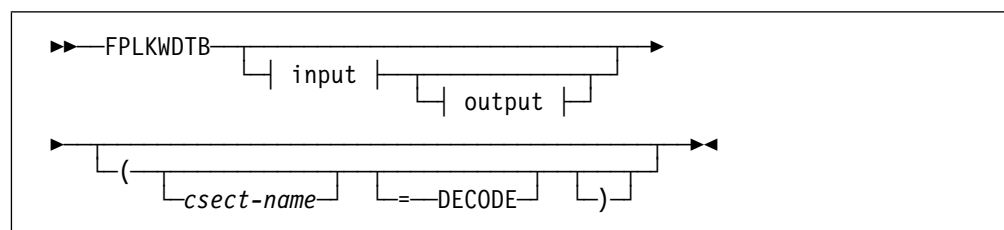
A source keyword table contains a line for each keyword and optionally comments. Comments begin with an asterisk and extend to the end of the line. Blank lines and lines that contain only comments are ignored. Case is ignored in the source keyword table. *CMS Pipelines* keywords are defined by blank-delimited words:

- The keyword identifier. This is a symbolic name under which a keyword is known to the code. The keyword identifier must be one or two characters. It is prefixed with the module name to obtain the label that will be the entry point for the keyword in the keyword table. Thus, the identifier must contain only characters that can be specified as an entry point in an Assembler control section. (That is, English alphanumerics and the three national use characters “#@\$”.)
- The keyword. This is the character string that is tested against an operand of a stage in a pipeline. Synonyms are specified by several lines that have the same identifier (the first word). The keywords are tested in the order they appear in the concatenated input files (the source keyword tables).
- A number specifying the minimum truncation that should be accepted for the keyword. If the second word contains more than eight characters, you must specify a minimum truncation count and it must be a number that is 8 or less. 0 (the default) specifies that no truncation should be accepted.

A sample line of a keyword table:

```
at assist 3
```

FPLKWDTB—Generate a Keyword Table Object Module



The FPLKWDTB command generates an object keyword table from one or more source keyword tables. It supports two blank-delimited words and options, which is specified after a left parenthesis:

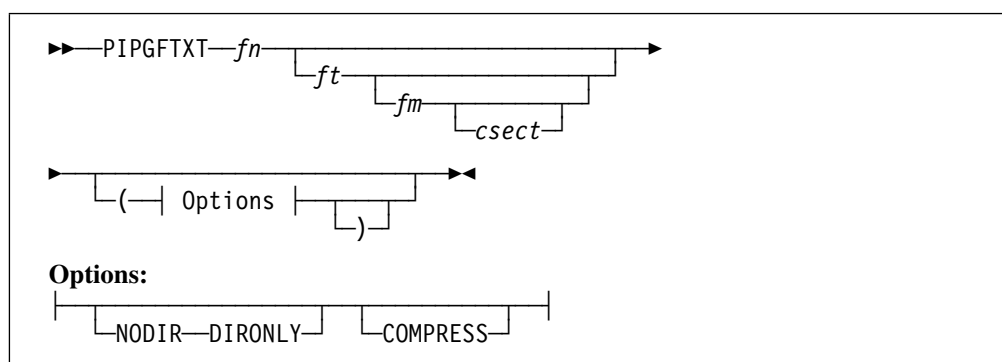
- Input file specification. The default input file is SYSTEM KWDTABLE on CMS; it is FPLKWD FPLPARMS on z/OS, the member FPLKWD of the data set allocated to FPLPARMS.
- The output file (which contains a single object module). By default, the output file is FPLKWD TEXT.
- The options field contains the ESD name of the control section that will contain the object keyword table and a keyword, which is specified after an equal sign.

Filter Packages

The default control section is the file name of the output file or its default.

When the option `DECODE` is specified, the object module also contains a table (in a format similar to an entry point table) that is used to decode a keyword to determine the identifier for the keyword. Because more than one keyword identifier is used for a particular keyword, this entry point table contains one to three identifiers rather than a pointer to an entry point. The `DECODE` option must be specified when generating the keyword table in the main pipeline module. Do not specify this option for a keyword table that is included in a type 1 filter package. When `decode` is specified for a type 2 filter package, the control section name must be specified as `FPLKWD`.

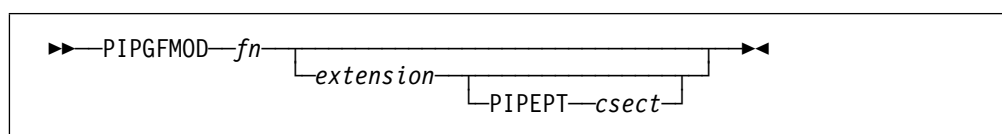
PIPGFTEXT—Generate Object Module from Program Directory



Use the `PIPGFTEXT EXEC` to create a filter package (TEXT file) from a pipeline filter package description file (input file) containing either REXX or Assembler user written *CMS Pipelines* programs. The input file consists of one or more records, and each record contains fields describing various aspects of the stage. `PIPGFTEXT` also generates an entry point table as part of the filter package, unless you specify the `NODIR` option. The `DIRONLY` option generates only the entry point table. The name of each REXX program is its entry point. An Assembler program can have multiple entry points. Specify the name of the Assembler program on the `PIPDESC` assembler macro.

The filter package (TEXT file) is created with the same name specified by the `FN` operand.

PIPGFMOD—Generate Filter package Load Module



Programs

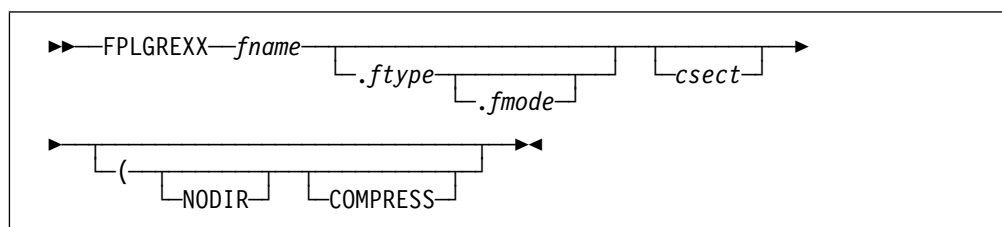
The programs must be in the format of object modules. For Assembler programs, this is clearly the output from the Assembler.

REXX programs can be processed in two different ways:

- The `FPLGREXX` command (which is used by `FPLGRMOD`) builds an object module that contains the lines of the REXX program. This program is passed to the interpreter when it is called. If the program is in the alternate format (compiled with the `CEXEC` option), the program is run by the runtime environment it specifies.

Specify the NODIR option if you will be supplying the entry point table for the filter package, for example, if it will contain compiled REXX programs as well as interpreted ones.

- The REXX compiler generates an object module when the option OBJECT is used.



Generating a Sample Type-1 Filter Package

Given this REXX program stored in the file MYFILTER REXX:

```
/* My very own filter.                                     */
parse source s
'output This is my very own filter:' s
exit RC
```

Perform these steps:

1. Compile the REXX program:


```
rexxc myfilter rexx (object nocexec
```
2. This PIPE command builds an entry point table that specifies that the entry point SAMPFILT will call the REXX program just compiled:


```
pipe literal sampfilt myfilter | > sample eptable a
```

►Ready;

You may instead wish to use XEDIT to create the source entry point table.

3. Generate the object module containing the entry point table:


```
fpleptbl sample sampept.text ( pipept
```

►Ready;

4. Point at the object module library that contains the glue code:


```
global txtlib fpluser
```

►Ready;

This step is required only for the “Runtime Library Distribution” of *CMS Pipelines*; skip this step with VM/ESA.

5. Load the glue code:


```
load pipnxf (rldsava noauto
```

►Ready;

(On VM/ESA, use DMSFPF for the name of the module to be loaded.) Ignore any messages that PIPEPT is undefined. This step must be done separately to ensure that the glue module is first in the generated module. RLDSAVE specifies that the module should be relocatable. NOAUTO specifies that CMS should not search for PIPEPT TEXT.

6. Include the other object modules:


```
include sampept myfilter
```

►Ready;

Filter Packages

You may wish to inspect the load map at this point.

7. Make a module file:

```
genmod sampfp (from pipnxf
►Ready;
```

The FROM option is *de rigueur*. (Specify the option from dmsfpf on VM/ESA.)

8. Remove any existing filter package from storage:

```
nucxdrop sampfp
```

If the filter package is one that is installed automatically (for example, PIPUSERF), the nucleus extension has a leading asterisk (to reduce the risk of accidentally invoking the package as a command). Thus, you might drop the nucleus extension *pipuser.

9. Install the filter package:

```
sampfp
►Ready;
```

If the filter package is one that is installed automatically (for example, PIPUSERF), you should issue the command PIPMOD INSTALL to make the main module install the filter package.

10. Test the filter:

```
pipe sampfilt | console
►This is my very own filter: CMS COMMAND SAMPFILT REXX * sampfilt ?
►Ready;
```

Appendix F. Pipeline Compatibility and Portability between CMS and TSO

CMS Pipelines is implemented in a layered fashion. The bulk of the *CMS Pipelines* code uses only *CMS Pipelines* services. This part does not depend on any particular operating system; it will run on any processor that is supported by the operating system.

Some modules determine the operating environment at run time, selecting the appropriate path dynamically. A few device driver modules are specific to the CMS or z/OS environment.

Level 1.1.9 of *TSO Pipelines* is incorporated into BatchPipes/MVS Release 2 (IBM Program Number 5655-065) under the name of BatchPipeWorks™. This product is not being enhanced.

TSO Commands Supplied with *TSO Pipelines*

The pipeline module FPLPIPE is linked with seven aliases: FPLDEBUG, FPLHLASX, FPLMVATT, FPLRESET, FPLRXSC, FPLUNIX, and PIPE. The aliases FPLHLASX, FPLMVATT, and FPLRXSC are reserved for *TSO Pipelines* internal use; program checks are likely if they are invoked as TSO commands.

FPLRESET

This command is equivalent to the CMS command “NUCXDROP PIPMOD”. It causes *TSO Pipelines* to release any resources and storage it may have acquired. Do not issue FPLRESET while any pipelines are running.

FPLDEBUG

This command is not intended for general use. It verifies the *TSO Pipelines* global area and displays information that may be helpful in isolating a problem.

When the pipeline environment (its control blocks) is correct, FPLDEBUG will just display the address of the global area.

```
fpldebug
PLMVS1224I TSO Pipelines global area is at 16D01628.
READY
```

In addition, FPLDEBUG may issue messages identifying tasks it knows about. The information displayed includes the contents of the TCB tokens. Note that the fact that *TSO Pipelines* knows about a task does not mean that the task is still active; most likely the task represents the last TSO command, which has terminated by the time you can issue the FPLDEBUG command.

CMS/TSO Compatibility and Portability

```
pipe q
FPLINX086I CMS/TSO Pipelines, 5654-030/5655-A17 1.0111 (Version.Release/Mod) -
Generated 21 Feb 2003 at 15:46:43
READY
fpldebug
FPLMVS1224I TSO Pipelines global area is at 1FB012B8
FPLMVS1237I Active process and thread IDs:00000948 0000004D 00000036 008A5AB8(
hexadecimal)
READY
```

FPLUNIX

Is the entry point for running the PIPE command in the UNIX System Services environment; that is, from the shell. See the following section.

Using the PIPE Command from Unix System Services

TSO Pipelines supports running under the USS shell. For this to work, you need to create an external link to the FPLUNIX entry point:

```
erwxrwxrwx 1 CCJOHN MKTGRP 7 Jul 24 1999 pipe -> FPLUNIX
```

Note in particular that the external link contains the member name rather than anything else you might think it should contain.

In addition, the STEPLIB environment variable must be set if the module is not in link pack:

```
CCJOHN:/home/ccjohn: >echo $STEPLIB
CCJOHN.TSO.LOAD
CCJOHN:/home/ccjohn: >pipe q
FPLINX086I CMS/TSO Pipelines, 5654-030/5655-A17 1.0111 (Version.Re
lease/Mod) - Generated 15 Feb 2000 at 12:28:57
```

TSO Pipelines writes error messages to standard error (file descriptor 2). *console* reads from standard input (file descriptor 0) and writes to standard output (file descriptor 1). In addition, the conveniences `stdin`, `stdout`, and `stderr` are available.

Be sure to quote your pipeline specifications:

```
CCJOHN:/home/ccjohn: >pipe literal abc|cons
cons: FSUM7351 not found
CCJOHN:/home/ccjohn: >pipe "literal abc|cons"
abc
```

In the first command the pipe character was interpreted by the shell; the PIPE command saw only the *literal* stage.

Pipeline Specifications—The PIPE Command

The pipeline specification parser does not depend on the operating system; a pipeline specification is scanned in the same way on CMS and on TSO.

From a REXX program (EXEC) on TSO, issue the pipeline specification may be addressed to several environments.

From a normal TSO REXX program (the merged environment, as it is called), you may address these environments: TSO, LINK, or ATTACH. From a stage written in REXX, can address only LINK or ATTACH, but you can issue TSO commands using *command* or *tso*. Address LINK is required when issuing multiple PIPE commands that must run in the same unit of work under DB2.

On TSO, REXX filters are resolved from partitioned data sets. The CMS file name (first word) corresponds to the member name; the CMS file type (second word) specifies the DDNAME of the data set. DDNAME=FPLREXX is the default.

Built-in Programs

Only device drivers and host command interfaces depend on the operating system; all filters and gateways are available both on CMS and on TSO.

Device Drivers and Host Command Interfaces Supported Identically on CMS and on TSO

These device drivers are available in both environments and perform the same function: *browse*, *emsg*, *hole*, *hfs*, *hfsdirectory*, *hfsquery*, *hfsreplace*, *hfsstate*, *hfsxecute*, *immcmd*, *ispf*, *literal*, *subcom*, *tcpclient*, *tcpdata*, *tcplisten*, *timestamp*, *udp*, and *3277enc*.

Device Drivers and Host Command Interfaces Supported on CMS only

These device drivers are supported only on CMS: *afffst*, *cms*, *cp*, *diskback*, *diskrandom*, *diskupdate*, *ldrtbls*, *mdiskblk*, *nucext*, *reader*, *starmsg*, *starmon*, *starsys*, *statew*, *tape*, *uro*, *vmc*, *xab*, *xedit*, and *xmsg*. In addition all device drivers for SFS files are supported on CMS only.

Device Drivers and Host Command Interfaces Supported on TSO only

The device drivers in Figure 410 are available with *TSO Pipelines* only. The fourth column shows the level at which the support was incorporated.

Figure 410. Summary of TSO Only Device Drivers		
Program	Description	L
<i>listcat</i>	Provide data set names that are qualified by a specified qualifier.	9
<i>listdsi</i>	Provide detailed information about data sets.	9
<i>listispf</i>	Reads the directory of a PDS into the pipeline, formatting the user data if it was stored by ISPF. The output can be limited to information about selected members.	7
<i>mjsc</i>	Send commands to a WebSphere MQ queue manager.	11
<i>sysdsn</i>	Test if data sets exist.	9
<i>sysout</i>	Write a SYSOUT data set.	9
<i>sysvar</i>	Write the contents of system variables into the pipeline.	7
<i>tso</i>	Issue TSO commands and trap the response into the pipeline. The individual return codes are written to the secondary output stream, if one is defined.	7
<i>writepds</i>	Replace members of a PDS. ISPF status can be maintained.	7

CMS/TSO Compatibility and Portability

Device Drivers and Host Command Interfaces Supported Differently on CMS and on TSO

Figure 411 lists the device drivers that are available both on CMS and on TSO, but are not entirely compatible. The first column shows the program name(s). The second and third columns contain remarks about the differences between the two implementations. The fourth column shows the level of the *TSO Pipelines* at which the support was added.

Figure 411 (Page 1 of 2). Differences Between CMS and TSO

Program	CMS	TSO (MVS)	L
< > >>	CMS file name specifications are used. Output data sets are created if they do not already exist.	z/OS DSNAME or DDNAME specifications are used. Generation data groups are supported. Output data sets written by DSNAME must be allocated and cataloged. Only the primary stream is supported.	7
<i>command</i>		Does not intercept the response.	6
<i>console</i>		Only EOF and NOEOF are supported.	6
<i>delay</i>		At most 16 <i>delay</i> stages can be active concurrently.	7
<i>fullscr</i>		Only NOREAD, CONDREAD, and READFULL are supported. <i>fullscr</i> works with the TSO terminal only.	7
<i>fullscrq</i> <i>fullscrs</i>		No arguments are allowed; only the TSO terminal is supported.	7
<i>getfiles</i>	Input lines should contain file names or data set specifications that are appropriate to the < device driver. If it is present, the string “&1 &2 ” is removed from columns 1 to 7 of the record.		7
<i>help</i>		Uses <i>browse</i> to display the help file. It defaults to TSO help if it cannot find the pipeline help library.	7
<i>listpds</i>	Files are specified with three words: file name, type, and mode.	The data set name or DDNAME is specified as a single word.	7
<i>members</i>	Files are specified with three words: file name, type, and mode. Always supply the member names on the input stream to make these as compatible as possible.	A single word specifying the DSNAME or DDNAME is followed by the list of members to read. <i>readpds</i> is a synonym.	7
<i>pdsdirect</i>	Reads the first record and the directory records without unblocking them. Files are specified with three words: file name, type, and mode.	Synonym for <i>listpds</i> . The data set name or DDNAME is specified as a single word.	7
<i>printmc</i>		A synonym for <i>sysout</i> , which writes a SYSOUT data set that has machine carriage control (SYSOUT=A). The output class and the output definition can be specified.	8

Figure 411 (Page 2 of 2). Differences Between CMS and TSO

Program	CMS	TSO (MVS)	L
<i>punch</i>		A synonym for <i>sysout</i> , which writes a SYSOUT data set that does not have carriage control (SYSOUT=B). The output class and the output definition can be specified.	8
<i>rexx</i>		REXX programs run in reentrant environments, which are not merged with TSO.	6
<i>rexxvars</i>		No environments can be reached beyond the one applying to the PIPE command.	6
<i>sql</i>		Insert on a cursor is not supported by DB2. The subsystem name is specified as an option.	6
<i>sqlcodes</i>		The memory of these codes is lost when the PIPE command terminates.	6
<i>stack</i>	When first in the pipeline, <i>stack</i> empties the program stack but does not read lines on the external input queue.	When first in the pipeline, <i>stack</i> empties the program stack and the external data queue. <i>stack</i> uses the IRXSTK default service.	7
<i>state</i>		Only the data set name is written to the primary output stream. Allocations can be queried. The existence of a data set does not imply that the user has read authority; nor does it imply that the data set is immediately available (it could be migrated).	7
<i>stem</i>		No environments can be reached beyond the one applying to the PIPE command.	6
<i>storage</i>		The third operand is ignored, but it must still be specified with correct syntax. When <i>storage</i> is not first in a pipeline, it does not change the PSW protection key when it copies a record into storage (it cannot, being an unauthorised program).	6
<i>var</i> <i>vardrop</i> <i>varfetch</i> <i>varload</i> <i>varset</i>		No environments can be reached beyond the one applying to the PIPE command.	6

CMS/TSO Compatibility and Portability

REXX Filters

Programs that do not use the Address instruction and do not rely on external functions are directly transportable between the two environments; they should work without change if they do not contain pipeline specifications that have incompatible device drivers.

Consider using CALLPIPE instead of Address command pipe; the results are the same for correct pipelines.

When incompatible device drivers are used, you must use the Parse Source instruction to determine the environment in which the program runs:

```
/* Dual-path for TSO and CMS */
parse source where . my_fname my_ftype my_fmode . env .
If where='TSO'
    Then dsn='names.text'
    Else dsn=userid() 'names a'
'callpipe <' dsn '|...
```

Assembler Programs

Programs should be 31-bit capable and should use *CMS Pipelines* services rather than services of an operating system. The interface defined in *CMS Pipelines Toolsmith's Guide and Filter Programming Reference*, SL26-0020 remains supported. The PIPEPVR macro is now required. The macro library is shipped as part of *TSO Pipelines*.

Filter Packages

TSO Pipelines supports packages managed with *filterpack*.

Appendix G. Format of Output Records from *runpipe* EVENTS

runpipe EVENTS produces a detailed trace of the execution of a pipeline set in a format that is suitable for processing by a program that runs concurrently with the pipeline set.

The event record contains an 8-byte common prefix which is followed by variants that define the individual record types.

Record Prefix

The format of the first eight bytes is common to all event records.

Offs	Len	Description
0	1	Record type. This byte defines which particular variant record is present. The variant data (if any) begin at offset 8.
1	1	X'00' to indicate the format specified here. A nonzero value would indicate that the record is extended or contains different information.
2	2	Reserved.
4	4	Reference. A unique number that defines a particular stage, pipeline specification, or pipeline set. This number is unique across all concurrent stages, pipeline specifications, and pipeline sets. The reference for a stage or a pipeline specification is unique until the pipeline specification ends. The reference for the pipeline set is unique for the particular input record which is being processed by <i>runpipe</i> . Thus, the references are not necessarily unique over the entire CMS session; they might be reused.
8	v	Beginning of record variations.

00—Message

This record is written when a message is issued. The message is suppressed.

Offs	Len	Description
0	1	X'00' for Message.
4	4	The reference for the issuing stage, if known. This field contains binary zeros when the stage is not known.
8	4	Zeros or the address of a character field containing the module name.
12	4	The message number. If negative, substitution is a list; if positive, a single item is substituted.
16	8	Substitution item if the message number is positive.
16	4	Address of substitution list if the message number is negative.
20	4	Number of items in substitution list.
24	1	Number of bytes of substituted message text.
25	n	Substituted message text.

Event Records

01—Begin Pipeline Set

Offs	Len	Description
0	1	X'01' for Begin Pipeline Set.
4	4	Reference for the pipeline set.
8	4	The address of the initial pipeline specification.
12	4	The length of the initial pipeline specification.

02—End Pipeline Set

Offs	Len	Description
0	1	X'02' for End Pipeline Set.
4	4	Reference for the pipeline set.
8	4	Return code.

03—Enter Scanner

This record has no variant data. The pipeline specification is described in a set of event records consisting of one for the overall pipeline specification and a record for each pipeline, stage, connector, and label reference. The scanner end record identifies the end of the pipeline specification.

Offs	Len	Description
0	1	X'03' for Enter Scanner.
4	4	Reference for the pipeline set.

04—Pipeline Vector Allocated

This record contains the PIPSCBLK data area.

Offs	Len	Description
0	1	X'04' for Pipeline Vector Allocated.
4	4	Reference for the pipeline specification.
8	8	pipedef (one trailing blank). A magic token.
16	1	The level of the scanner records. 01 Initial specification. This record is 40 bytes long. The last two bytes contain zeros. 02 The record is 88 bytes long.

Offs	Len	Description
17	1	Flag byte. When the flag byte contains zeros, the pipeline specification was issued by PIPE or <i>runpipe</i> . 01 Function is CALLPIPE 02 Function is ADDPIPE When both bits are on, the pipeline specification is scanned after the CALLPIPE rules, but connected after the ADDPIPE rules.
18	1	Option flag byte. 01 option TRACE 02 option LISTERR 04 option LISTRC 10 option STOP 80 option LISTCMD
19	1	Reserved.
20	8	The option NAME. The value is truncated after eight bytes. See below for a pointer to the complete name.
28	2	Bits to turn on in the rightmost halfword of the message level.
30	2	Bits to clear in the rightmost halfword of the message level.
32	1	The stage separator character. This byte contains X'00' when the pipeline specification is issued as an encoded pipeline specification.
33	1	The end character. This byte contains a blank when no end character is defined. It contains X'00' when the pipeline specification is issued as an encoded pipeline specification.
34	1	The escape character. This byte contains a blank when no escape character is defined. It contains X'00' when the pipeline specification is issued as an encoded pipeline specification.
35	3	Reserved.
38	2	Offset to the address of the list of stage definitions. This offset is zero in the first level of the block.
40	4	Address of the original pipeline specification (when the pipeline specification was issued from a command string) or zero for an encoded pipeline specification for which there is no original string.
44	4	The length of the original pipeline specification string.
48	4	The address of the name for the pipeline specification.
52	4	The length of the name. This length may be greater than 8.
56	32	Reserved. Contains zeros.

Event Records

05—Leave Scanner

The pipeline specification is either abandoned or handed over to the pipeline dispatcher to run.

Offs	Len	Description
0	1	X'05' for Leave Scanner.
4	4	Reference for the pipeline specification.
8	4	Return code. The pipeline specification is abandoned if this return code is not zero.

06—Scanner Item

The scanner items describe the beginning of a pipeline, a stage, a label reference, or a connector. The first scanner item is a pipeline begin variant. This variant of the event record contains the PIPSCSTG data area; it has four variants.

Offs	Len	Description
0	1	X'06' for Scanner Item.
4	4	Reference to the pipeline specification, except for record variant 01 (stage) where this defines the reference for the stage.
8	4	Pipeline number. The first pipeline has number 1.
12	4	Stage number. This field is zero in the pipeline begin item.
16	2	Number of bytes in the remainder of the record.
18	1	Item type. This specifies the variant of the remaining data in the record.
Variant for pipeline begin.		
18	1	X'00' Beginning of a pipeline.
19	1	Reserved.
Variant for stage.		
18	1	X'01' Stage.
19	1	Option flag byte. 01 option TRACE 02 option LISTERR 04 option LISTRC 10 option STOP 80 option LISTCMD
20	2	Bits to turn on in the rightmost halfword of the message level.
22	2	Bits to clear in the rightmost halfword of the message level.
24	8	Label or blank.
32	4	Stream identifier or zeros.
36	4	Entry point address or zero to resolve the entry point from the verb.

Offs	Len	Description
40	4	Address of verb.
44	4	Length of verb string. The length of the verb string is zero when <i>rexx</i> is implied and in some other cases. The length includes a trailing blank; it might include leading blanks.
48	4	Address of parameter string.
52	4	Length of parameter string.
Variant for label.		
18	1	X'02' Label reference.
19	1	Reserved.
20	4	Reference number for the stage being referenced.
24	8	The label being referenced.
32	4	Stream identifier or zeros.
Variant for connector.		
18	1	X'03' Connector.
19	1	Flag. 01 Input. 02 Output. 04 Conditional. Connect only if the referenced stream exists. This flag is supported only for encoded pipeline specifications.
20	4	Stream identifier or zeros.
24	4	Stream number.

07—Calling Syntax Exit

The pipeline specification parser's third pass is about to call a syntax exit. No record is produced to indicate the return from the exit. If records of this type are produced, a nonzero return code in record type 05 indicates that one or more syntax exits returned with a nonzero return code.

Offs	Len	Description
0	1	X'07' for Calling Syntax Exit.
4	4	Reference for the stage.
8	4	Address of work area allocated for stage.
12	4	Register 1. Depending on the programming language, this may be a pointer the the stage's work area or to a parameter list.
16	48	Registers 0 through 11.

Event Records

08—Start Stage

The pipeline dispatcher is about to call the initial entry point for a stage.

This variant has no additional data.

Offs	Len	Description
0	1	X'08' for Start Stage.
4	4	Reference for the stage.

09—End Stage

The stage returns to the pipeline dispatcher.

Offs	Len	Description
0	1	X'09' for End Stage.
4	4	Reference for the stage.
8	4	Return code.
12	4	Size of the work area allocated (in doublewords).
16	4	Highwater mark of other allocated storage (in doublewords).

0A—Resuming Stage

The pipeline dispatcher is about to return control to a stage. The stage has previously called a pipeline dispatcher service.

Offs	Len	Description
0	1	X'0A' for Resuming Stage.
4	4	Reference for the stage.
8	4	Return code.
12	4	Register 0.
16	4	Register 1.
20	2	Reserved.
22	2	Two-character service code for the previous service call issued by the stage. Refer to the description of record type 0B.
24	4	Number of the input stream that is currently selected by the stage.
28	4	0 if the currently selected input stream is not connected. -1 if the currently selected input stream is not selected by the other end of the connection. If positive, the reference to the producer stage.
32	4	Stream number selected by producer. -1 when the previous field is not positive.
36	4	Number of the output stream that is currently selected by the stage.

Offs	Len	Description
40	4	0 if the currently selected output stream is not connected. -1 if the currently selected output stream is not selected by the other end of the connection. If positive, the reference to the consumer stage.
44	4	Stream number selected by consumer. -1 when the previous field is not positive.

0B—Calling Dispatcher Service

A stage issues a service request to the pipeline dispatcher.

Offs	Len	Description
0	1	X'0B' for Calling Dispatcher Service.
4	4	Reference for the stage.
8	4	Reserved.
12	4	Register 0.
16	4	Register 1.
20	2	Reserved.
22	2	Two-character service code. See below.

The service codes are:

- CM Issue Pipeline Command. Register 1 contains the address of the command; register 0 contains the length of the command.
- CT Commit. Register 0 contains the commit level requested.
- ES The stage returns on the initial call from the dispatcher. An end stage variant (09) follows.
- LO PEEKTO.
- MS Miscellaneous services. The value in register 0 defines the service requested:
- 0 Stall the pipeline specification.
 - 1 Suspend.
 - 2 Set break wait flag.
 - 3 Clear break wait flag.
 - 4 Process an encoded pipeline specification. This is used to issue ADDPIPE and CALLPIPE requests that are assembled into built-in programs.
 - 5 Test if events are enabled. Register 1 contains the address of a fullword into which the dispatcher stores the stage's reference number.
- PA Set parameter registers to their original values on entry to stage. (Not used.)
- PI READTO. Register 1 contains the address of the buffer; register 0 contains the length of the buffer (zero means release input record).
- PO OUTPUT. Register 1 contains the address of the record; register 0 contains the length of the record. When the stage is resumed, register 0 contains the number of bytes consumed. This is reflected in the next record with type 0A.
- SA STREAMSTATE. Register 0 contains the encoding of the side to test:

Event Records

- 1 Input.
- 2 Output.

Register 1 contains the stream number or stream identifier (if the leftmost byte is nonzero).

SH SHORT.

SL SELECT. Register 0 contains the encoding of the side to select:

- 1 Input.
- 2 Output.
- 3 Both.
- 4 Any input.

Register 1 contains the stream number or stream identifier (if the leftmost byte is nonzero).

SN STAGENUMBER.

SV SEVER. Register 0 contains the encoding of the side to sever:

- 1 Input.
- 2 Output.

WE WAITECB. Register 0 contains the code to post when an input record arrives (if the break wait flag is enabled). Register 1 contains the address of the PIPECB.

X Set dispatcher exit. Register 0 contains the address of the exit routine or zeros.

0C—Pipeline is Stalled

This event has no variant data. A record with code X'0D' is written for each stage.

Offs	Len	Description
0	1	X'0C' for Pipeline is Stalled.
4	4	Reference for the pipeline set.

0D—State of Stage

Offs	Len	Description
0	1	X'0D' for State of Stage.
4	4	Reference for the stage.
8	4	Encoded state of stage.
12	8	Decoded state of stage.

0E—Pipeline Committing

Offs	Len	Description
0	1	X'0E' for Pipeline Committing.
4	4	Reference for the pipeline specification.
8	4	The committed aggregate return code.
12	4	The level to which the pipeline is committed.

0F—Console Input

A *console* stage that is first in a pipeline requires a record. The *console* stage that runs under control of *runpipe* EVENTS does not read from the terminal; rather, it signals a console input event. A stage that processes the event records without their being delayed from the *runpipe* stage can store an input record in the input buffer and set the number of bytes in the feedback word. If this event record is consumed without a record being stored, the *console* stage assumes a null record was entered.

Offs	Len	Description
0	1	X'0F' for Console Input.
4	4	Reference for the <i>console</i> stage.
8	4	Address of the input buffer.
12	4	Length of the input buffer. The controlling stage must at most store this number of bytes in the buffer.
16	4	Address of a fullword into which the controlling stage should store the actual length of the input line. (The feedback word.)

10—Console Output

A *console* stage that is not first in a pipeline has read an input record. The *console* stage that runs under control of *runpipe* EVENTS does not write to the terminal; rather, it signals a console output event. A stage that processes the event records without their being delayed from the *runpipe* stage can obtain the record from the buffer.

Offs	Len	Description
0	1	X'10' for Console Output.
4	4	Reference for the <i>console</i> stage.
8	4	Address of the output buffer.
12	4	Length of the output buffer.

11—Pause

pause has read an input record. A stage that processes the event records without their being delayed from the *runpipe* stage can now react to this event. The *pause* stage is resumed as soon as this record is consumed. There are no variant data for this event.

Offs	Len	Description
0	1	X'11' for Pause.
4	4	Reference for the <i>pause</i> stage.

Event Records

12—Subroutine Pipeline Complete

This record is generated when all stages of a subroutine have terminated and all connections are either at end-of-file or restored to their original state.

Offs	Len	Description
0	1	X'12' for Subroutine Pipeline Complete.
4	4	Reference for the pipeline specification.
8	4	The return code.

13—Caller is Waiting for Subroutine Pipeline to Complete

This record is generated when a stages has issued the CALLPIPE pipeline command. A type 12 record is generated when the stage is again ready to be dispatched.

Offs	Len	Description
0	1	X'13' for Caller is Waiting for Subroutine Pipeline to Complete.
4	4	Reference for the stage that issued the CALLPIPE pipeline command.
8	4	The reference value for the pipeline specification that the stage is now waiting on.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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 grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (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

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 jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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 Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

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. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/VM.

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 IBM copyright and trademark information - United States (www.ibm.com/legal/us/en/copytrade.shtml).

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM & website..

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein. IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED “AS-IS” AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

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 Online Privacy Statement Highlights at <http://www.ibm.com/privacy> and the IBM Online Privacy Statement at <http://www.ibm.com/privacy/details> in the section entitled "Cookies, Web Beacons and Other Technologies", and the IBM Software Products and Software-as-a-Service Privacy Statement at <http://www.ibm.com/software/info/product-privacy>.

Explanation of Terms

Glossary

A

Abut. To put two things together with nothing between them. For instance, two abutted strings are put together without separating blanks.

Argument string. Characters coded after the name of a stage. The argument string is made available to the stage.

B

Backus-Naur form. (BNF) A notation for syntax definition, invented in the early sixties. The dialect used in *CMS Pipelines* manuals is described in Chapter 20, "Syntax Notation" on page 222.

Block descriptor word. (BDW) z/OS term. The BDW is a fullword prefix to a block in variable format. It contains the total length of the block in the first halfword and zeros in the second.

Blocked. When the pipeline dispatcher has blocked a stage, the stage is not run. A stage is blocked when it accesses a connection if the other side of the connection is not prepared to read or write a record.

Buffer. A stage that reads all its input before writing any output. Such a stage may be needed to ensure that a multi-stream pipeline does not stall. *buffer* and *sort* buffer the file; *lookup* buffers the secondary input stream.

C

Card. See *Punched card*.

Carriage control. Vertical skipping and spacing of line printers is controlled by commands that are normally stored in the first column of a print line. There are two types of carriage control, machine and ASA. Machine carriage control is the CCW command code that is to be used to print the line, whereas ASA carriage control consists of the characters 1, space, 0, -, and +.

Cascade. A sequence of filters to perform a function by each doing its little bit of the work. For instance, a cascade of *locate* stages finds lines containing all the strings specified in the cascade.

Channel Command Word. (CCW) An order to a System/370 channel to read or write a record, or to cause mechanical movement at the device.

CMS. Conversational Monitor System is the component of VM that provides timesharing facilities.

Collating sequence. Ordering of the character set used in a computer. When the binary encoding of one character is less than the encoding of some other character, then the first character is said to be before the second one in the collating sequence. Also used to designate all possible values for a single character. A byte contains 8 bits in IBM/370, so the collating sequence has 256 characters from X'00' to X'FF'.

Connection. A data path between two stages. A stage can have several input and output connections, but only one input and one output can transport data at a time. The active connection is changed by the select function.

Connector. An item at the beginning or end of a pipeline indicating how the pipeline is to be connected to streams in the stage defining the pipeline. A full connector consists of an asterisk, a period, a keyword indicating a direction, a period, a stream identifier, and a colon; it can be as short as *:.

Console. The terminal for the virtual machine. Also the name of a CMS macro that is used to access the terminal in full screen mode rather than in line mode.

Control stage. A stage that inspects the contents of a file and calls one of several subroutine pipelines to process the particular file format.

Coroutines. Programs being multiprogrammed in a way where programs explicitly transfer control amongst themselves.

CP. The Control Program component of VM manages the resources of a real computing system in such a way that multiple machines appear to exist. Each user on a VM system has a virtual machine.

D

Data stream. A data stream flows into a stage's input and out of its output. When pipelines intersect in a stage, the program can access more than one data stream. Streams are numbered from zero onward as the intersections are defined. A stream can be named with a stream identifier.

Device driver. A program that interfaces between the sequential data streams of *CMS Pipelines* and the world outside the pipeline. An example is *disk*, which reads and writes files on CMS minidisks.

Diagnose instruction. An interface between the virtual machine and CP.

Dispatcher. An operating system routine that selects work to be done and coordinates the execution of individual units of work.

DOS. (Disk Operating System.) A precursor of VSE.

Driver. Shorthand for device driver.

E

EDF. See Enhanced Disk Format.

End character. A character in a pipeline specification that separates pipelines. The stage to the left of an end character is a last stage; the one to the right of an end character is a first stage.

Enhanced Disk Format. (EDF) A CMS file storage format used on minidisks that are attached to the user's virtual machine. The minidisk is formatted into 512, 1K, 2K, or 4K physical blocks. See also "shared file system".

Enumerate. To list all instances. When matching, for instance in *split*, a string can be interpreted as a list of characters which must match successive positions in the input record, or as an *enumerated scalar* which matches a single character if it is any one of the characters in the string.

Escape character. A character in a pipeline specification that disables any special meaning of the following character.

EXEC. A command procedure normally written in the language REXX. It is stored in a file with file type EXEC. Commands in an EXEC are processed by CMS.

EXEC2. The command procedure language available in CMS prior to the introduction of REXX in VM/System Product Release 3.

F

F. Code indicating fixed record format. All records of a fixed file have the same length.

FB. Code indicating fixed blocked record format. Blocks in a FB file have a length that is a multiple of the record length.

File. A collection of records; an entity stored in a file system. Also used about the data that pass through a stage even though no physical file will exist.

Filter. A program using *CMS Pipelines* for data transport. Filters are stages that somehow transform the data passing through. Often, but not always, a filter completely processes one record before reading the next.

Filter package. Filter packages are separate module files with pipeline programs and message tables. Such modules are loaded as CMS nucleus extensions and identify themselves

to *CMS Pipelines*. A filter package can contain any mixture of programs written in REXX, PL/I, IBM C/370, or assembler.

First stage. The leftmost stage of a pipeline. A pipeline specification can contain several pipelines separated by end characters. A stage to the right of an end character is also a first stage.

File Status Table. (FST) Information about a file on a CMS minidisk is stored in the FST for the minidisk. This information includes the record format, the record length, and the date the file was written or appended to.

Flush. To empty a buffer.

Forms Control Buffer. (FCB) Control information to define where on a page a skip to a given channel should stop. A forms control buffer can be associated with a SPOOL file.

G

G. (Gigabyte.) =1024M =1,048,576K =1,073,741,824.

Gateway. A program used in a pipeline that is not a device driver or a filter.

Glue. A program that makes two other ones work together without modifying them. A glue module often transforms one data format or record layout to another.

H

Host. The operating system that *CMS Pipelines* runs on.

Host Interface. An interface to a CP, CMS, TSO, or z/OS function used by *CMS Pipelines*.

I

IEBCOPY. The name of a z/OS utility program used to transport data sets between systems.

Infrastructure. Collective name for service routines that provide services for the parts of a program that are related to performing the actual task as perceived by the user. Parameter scanning is such a routine.

Intersection. A stage that has access to more than one pipeline. The pipelines are said to intersect in that stage.

K

K. (Kilobyte.) =1024.

Explanation of Terms

L

Label. An identifier for a stage that defines multiple data streams, one for each occurrence of the label in a pipeline specification.

Landscape. Format for a command where all information is entered on a single line.

Last stage. A stage at the end of a pipeline specification. A stage to the left of an end character is also a last stage.

Left. Stages are ordered left to right such that a stage receives its input from the output of the stage to the left of it.

Line. Originally used for records printed on a printer or a terminal. *CMS Pipelines* uses this term interchangeably with record.

Line end character. A character in a file marking the end of one line and the beginning of another. The line end character is not considered part of either record. Line end characters are not used by CMS, except in OpenExtensions text files. CP uses X'15' as a line end character for responses returned to the virtual machine in a storage buffer, but this is not visible to the user since *CMS Pipelines* performs the deblocking.

Locate mode. Term for data management processing where the record is processed in a buffer allocated by the access method instead of a buffer allocated by, or in, the program. Data management provides the address and length of input records. The converse is move mode.

M

M. (Megabyte.) =1024K =1,048,576.

Message level. A number set by the command PIPMOD MSGLEVEL. The binary representation of this number is interpreted as a set of switches controlling the degree of additional checking performed, and the number of additional messages issued, if any.

Move mode. Term for data management processing where the record is processed in a buffer allocated by, or in, the program; data management moves the record to or from the user's buffer. The converse is locate mode.

MVS. (Multiple Virtual Storage.) An operating system for IBM System/390 mainframe computers.

N

Netdata. A blocking format used to transmit files between IBM systems. The format includes information about the file as well as the contents of the file.

Null. Empty; containing nothing; having zero length.

Null record. A record with no data; it has length zero. Such a record can indicate end-of-file.

Null stage. Two stage separators next to each other with only blanks between them; a stage separator next to an end character or at the end of the pipeline specification.

O

Operating System. (OS) IBM's operating system for the System/360 family of computers was called OS (OS/360 to be precise). It was a family of operating systems, which evolved into z/OS. A subset of the interfaces defined for this system are available under CMS.

P

Pipeline. A list of programs where the output of one is passed to the input of the next. Each program is written to use a simple interface that transports a sequential data stream.

Pipeline command. A command issued from a stage (in particular a REXX program) to the pipeline dispatcher. The pipeline topology is controlled by pipeline commands. In a REXX filter, the default command environment processes pipeline commands rather than CMS commands.

Pipeline dispatcher. The program that transfers control between stages to ensure an orderly flow of data through the pipeline.

Pipeline specification. The character string that defines a pipeline. Stages are separated by a special character, the stage separator, which is the solid vertical bar (|) by default.

Portrait. A command written over several lines is said to be in portrait format. The portrait format of a pipeline command typically has the specification of each stage on a separate line.

Primary data stream. Stream number 0.

Printer. A virtual device, simulated by CP, used to write virtual printer SPOOL files in the CP SPOOL system.

Public. SQL term meaning that a resource is available to everyone.

Punch. A virtual device, simulated by CP, used to write virtual punch SPOOL files in the CP SPOOL system.

Punched card. Archaic data storage medium where characters are represented by holes cut in a piece of cardboard. The most widely used format stores 80 characters. A deck of punched cards is simulated by CP as a SPOOL file with a record for each card. CP enforces a maximum record length of 80 bytes.

Q

Quietly. Jargon for “with error messages suppressed” or “without issuing error messages”.

R

Record. Unit of information transmitted as a whole. A line of a file.

Record descriptor word. (RDW) In z/OS, a RDW is a fullword describing a logical record with the aggregate length (including the RDW) in the first halfword and zeros in the second one. Also used to describe the halfword length that precedes a record in the CMS file system when the file is in variable format.

REXX. (Reformed EXtended eXecutor.) The name of a programming language, implemented in CMS by the System Product Interpreter. Also designates programs written in REXX where the host commands go to the pipeline and are used for data transport.

Right. Stages are ordered left to right such that a stage delivers its output to the input of the stage to the right of it.

S

Scope. Where an identifier is recognised.

Secondary data stream. Stream number 1.

Segment descriptor word. (SDW) z/OS term. A fullword describing part of a logical record. The first halfword contains the length of the segment including the SDW. The third byte contains the segmentation flags that define whether the segment is the first, the last, or an intermediate one. The last byte is zero.

Segmentation Flags. Flag bits that define which part of a logical record is in the present segment. z/OS uses a different encoding for variable spanned than it uses for the netdata format. The encoding specifies whether the segment is the first, the last, the only, or a middle segment of a record.

Sequential data stream. Informal way to express a data stream that is processed in a sequential fashion, one record at a time, without going back to previous data.

Sever. Terminate the use of a stream. All connections to streams are severed when a stage returns control on the original invocation from the pipeline dispatcher. A stage can sever a connection explicitly with the SEVER pipeline command.

Shared File System. (SFS.) A file system introduced in VM/System Product Release 6. To facilitate sharing of data without compromising data integrity, data are stored on mini-disks that are attached to a “server” virtual machine; the user cannot access the file pool minidisks directly.

Shell. A program that reads lines from the terminal and interprets them as commands. Also called a Terminal Monitor Program.

Short Circuit. A stage can connect an input stream and an output stream with a short circuit. Records then bypass the stage that has performed the short circuit operation. A stage can issue the SHORT command to short circuit the currently selected input and output stream without waiting. It can also issue CALLPIPE to perform this operation: the stage waits until end-of-file is reflected on the short circuit; the output stream is available for further output when the stage resumes.

Span. A record is spanned across blocks when the first part of the record is in one block, and the rest of it is in one or more other blocks.

SPOOL. (Simultaneous Peripheral Operations On Line.) A system for controlling unit record devices. Also used to refer to the data set that holds the data being SPOOLED.

Stage. A program in a pipeline specification.

Stage separator. The character (normally |) that separates stages in a pipeline specification.

Stall. A condition in a pipeline network where not all stages have completed but stages are interlocked in such a way that no stage can be run.

Stemmed Array. A collection of REXX compound variables having the same stem, and a numeric index. The variable with index zero (for instance array.0) contains the number of data variables in the array; data are stored in variables with positive index, starting at 1. Thus, the first variable is array.1, the second variable is array.2, and so on.

Stream. Informal name for a data stream.

Stream identifier. A symbolic reference to a data stream.

Subcommand Environment. A named environment to which commands can be addressed on CMS. Many programs set up a subcommand environment to process commands issued by REXX programs that are called as a result of a user command to the program establishing the subcommand environment.

Subroutine pipeline. A pipeline, defined by `CALLPIPE`, to process data. The stage waits until the subroutine pipeline is complete.

SVC. (SuperVisor Call.) An instruction causing a switch from the user program to the operating system.

SVC 202. Used in CMS to issue commands by name.

SVC 203. Used in CMS to issue functions by number.

T

Task. An independent unit of work in an operating system. Each pipeline stage can be considered a task. *CMS Pipelines* tasks run as coroutines because control is passed from one to another when the pipeline dispatcher is called to read or write a record.

Terminal Monitor Program. (TMP) Program that reads lines from the terminal and interprets them as commands. Also called a Shell.

Tertiary data stream. Stream number 2.

Token. A word delimited by blank characters, parentheses, or both. A command is said to be comprised of tokens.

Tokenise. Build a list of doubleword (eight bytes) tokens from a character string. How this is done depends on the *scanning rules*, defining which characters end a token. The CMS scanning rules are that blank characters and parentheses delimit tokens; parentheses are scanned as separate tokens.

Transparent. A stage is transparent for return codes from CMS when it returns with the return code it got from the host.

TSO. The “Time Sharing Option” for z/OS. This allows z/OS users interactive access to the facilities of the operating system. Over the years many products have moved between the TSO and the CMS platforms.

U

Unit record. A (virtual) punched card or printed line. Readers, printers, and punches process unit records, and are thus referred to as unit record devices.

Unlimited. Often means “within the virtual storage available”.

V

V. Variable format.

VB. Variable blocked format.

VBS. Variable blocked spanned format.

Verb. The name of the entry point for a stage.

X

XEDIT. The editor used to edit files on CMS. Its formal name is VM/System Product Editor.

XEDIT Macro. A program written in REXX (or EXEC2) that is called from XEDIT. Through a defined interface, the program can obtain data from XEDIT and issue XEDIT commands.

Bibliography

Where to Get z/VM Information

The current z/VM product documentation is available in IBM Knowledge Center - z/VM (www.ibm.com/support/knowledgecenter/SSB27U)

Additional References

- The data streams used by 3270 terminals are described in:

IBM 3270 Information Display System, Data Stream Programmer's Reference, GA23-0059.

- These manuals may be needed when using *ispf* to access ISPF tables:

Interactive System Productivity Facility Dialog Management Guide, SC34-4009. *Interactive System Productivity*

Facility Dialog Management Services and Examples, SC34-4010.

- This manual may be useful when using *sql*:

DB2 Server for VSE & VM Application Programming, SC09-2889.

- General references:

Bell Systems Technical Journal **57.6** (July-August 1978).

AT&T Bell Laboratories Technical Journal **63.8** (October 1984).

Stuart J. McRae, *CMS and UNIX™—What They Can Learn from Each Other* Document SYSTELL/SJM-84.1, Systems & Telecoms Limited [now defunct].

Brian W. Kernighan and P. J. Plauger, *Software Tools in Pascal* Addison-Wesley 1981 ISBN 0-201-10342-7.

J. P. Morrison, *Data Stream Linkage Mechanism* IBM Systems Journal **7.4**, 1978, G321-5081 (reprint).

Index

Special Characters

; 573

-
 Picture character 747

,
 Picture character 748

! 23

 OR character in *all* 295

/
 Picture character 748

.
 Picture character 748

field identifier 720, 196, 179

filter 2

filter package 924
 Type 1 924
 Type 2 924

first reading station 720

\$
 Picture character 747

*
 Picture character 748

*ACCOUNT 594, 595

*COPY 66, 483

*LOGREC 594, 595

*MONITOR 589

*MSG 592, 797

*MSGALL 592, 797

*SYMPTOM 594, 595

&
 AND character in *all* 295

Coroutines
 Glossary definition 952

Locate mode
 Glossary definition 954

Move mode
 Glossary definition 954

+
 Picture character 747

< built-in program 263
 Example of use 9, 10, 11, 12, 15, 17, 18, 29, 30, 36, 40,
 46, 63, 66, 78, 80

<*m*sk built-in program 264

<*m*vs built-in program 265

<*o*e built-in program 266

<*s*fs built-in program 267

<*s*fs*s*low built-in program 268

> built-in program 270
 Example of use 8, 14, 15, 17, 29, 30, 44, 78, 81

>> built-in program 278
 Example of use 78

>>*m*sk built-in program 279

>>*m*vs built-in program 281

>>*o*e built-in program 282

>>*s*fs built-in program 282

>>*s*fs*s*low built-in program 285

>*m*sk built-in program 271

>*m*vs built-in program 273

>*o*e built-in program 275

>*s*fs built-in program 276

-
 NOT-character in *all* 295

| 23
 OR character in *all* 296

Numerics

14
 Diagnose 542

3270*bfra* built-in program 715

3270DS Structured Field 417

3270*enc* built-in program 717

3277*bfra* built-in program 715

3277*enc* built-in program 717

370 accommodation 824

3800
 EXEC 35

3way built-in program 651

407 Accounting Machine 719

4224 417

4F
 Code point 23

500 character limit on REXX clause 243

5A
 Carriage control 535, 675

64*decode* built-in program 717

64*encode* built-in program 718

9
 Picture character 748

A

A8
 Diagnose 536, 537, 676

abbrev built-in program 287
 Example of use 84

Abut
 Glossary definition 952

ACCESS 30, 785

Access list entry token 211

Access register 211

Accounting machines 149

Index

- acigroup* built-in program 288
 - ADDDPIPE pipeline command 751
 - addrdw* built-in program 289
 - Address ATTACH 935
 - Address instruction 114
 - Address LINK 935
 - Address space 28
 - address space identification token 210
 - address spaces 210
 - Address TSO 935
 - ADDSTREAM pipeline command 752
 - adrspc* built-in program 290
 - ADRSAPCE 290
 - AFTER program option
 - chop* 326
 - Example of use 60
 - insert* 450
 - pick* 518
 - split* 580
 - afffst* built-in program 293
 - aggrc* built-in program 294
 - Aggregate 39
 - Aggregate return code 247
 - ahelp* built-in program 430, 428
 - AID 415
 - ALL
 - XEDIT Subcommand 44
 - all* built-in program 295
 - Example of use 53
 - All Points Addressable printer 350, 508, 512
 - ALLDIRS
 - REXX 471
 - alserv* built-in program 296
 - ALSERV 296
 - ANYCASE program option 57, 128
 - abbrev* 287
 - between* 309
 - change* 323
 - chop* 326
 - collate* 332
 - inside* 450
 - joincont* 460
 - locate* 472
 - lookup* 475
 - merge* 498
 - nlocate* 501
 - notinside* 505
 - outside* 508
 - pick* 518
 - sort* 567
 - spill* 577
 - split* 580
 - strasmfind* 609
 - strasmnfind* 610
 - strfind* 611
 - strfrlabel* 612
 - ANYCASE program option (*continued*)
 - strip* 613
 - strnfind* 616
 - strtolabel* 617
 - structure* 618
 - strwhilelabel* 625
 - unique* 668
 - verify* 692
 - wildcard* 701
 - ANYOF program option
 - chop* 326
 - Example of use 59, 68
 - joincont* 460
 - locate* 472
 - nlocate* 501
 - space* 569
 - spill* 577
 - split* 580
 - strip* 613
 - APA 350, 508, 512
 - apldecode* built-in program 298
 - aplencode* built-in program 299
 - append* built-in program 300
 - Example of use 37, 38, 440
 - Arcane 253
 - Argument string
 - Glossary definition 952
 - ARIRVSTC
 - TEXT 800, 801
 - ASA carriage control 152
 - asatmc* built-in program 302
 - asmcont* built-in program 303
 - Example of use 67, 68, 69
 - asmfind* built-in program 304
 - asmnfind* built-in program 306
 - ASMSQL
 - DMSPQI 585
 - asmxpnd* built-in program 307
 - Assignment operator 179
 - ASYNCMS
 - REXX 163
 - ATTACH
 - Address 935
 - Attention ID 415
- ## B
- B
 - Picture character 748
 - Backus-Naur form
 - Glossary definition 952
 - Backwards propagation of end-of-file 90
 - beat* built-in program 308
 - BEGOUTPUT pipeline command 752
 - between* built-in program 309
 - Example of use 55

- bfs* built-in program 430
 - bfsdirectory* built-in program 431
 - bfsquery* built-in program 432
 - bfsreplace* built-in program 433
 - bfsstate* built-in program 434
 - bfsxecute* built-in program 435
 - Binding the plan 138
 - bit* syntax variable 224
 - polish* 527
 - BLANK program option 227
 - overlay* 510
 - pad* 514
 - split* 580
 - strip* 613
 - wildcard* 701
 - blank* syntax variable 224
 - block* built-in program 310, 275, 282
 - Example of use 124
 - Block descriptor word
 - Glossary definition 952
 - Blocked
 - Glossary definition 952
 - BOTHDISK
 - REXX 670
 - Bottom up parsing 182
 - BPXYERNO 837
 - BPXYERNO macro 837
 - BPXYSTAT 434
 - BPXYUSTN 433
 - BREAK program option 196
 - Break key 417
 - Breaks 195
 - browse* built-in program 315
 - brw* built-in program 315, 315
 - Buffer
 - Glossary definition 952
 - buffer* built-in program 317
 - Example of use 81, 82, 87
 - buildscr* built-in program 319
 - Byte streams 365
 - BYTES program option 345
 - drop* 381
 - take* 633
- C**
- c14to38* built-in program 350
 - C2X
 - REXX 212
 - CAF OPEN 801
 - CALLPIPE 246
 - CALLPIPE pipeline command 753, 246
 - Card
 - Glossary definition 952
 - Card file 149
 - Carriage 150
 - Carriage control
 - 5A 535, 675
 - Glossary definition 952
 - Carriage control character 150
 - Carriage tape 150
 - Cascade
 - Glossary definition 952
 - Case, ignoring 128
 - casei* built-in program 322
 - Example of use 57
 - Caseless 57
 - CATTWO
 - REXX 388
 - PRINT 919
 - CCW 150
 - CD System keyword 795
 - CEXEC 547
 - CHANGE
 - XEDIT Subcommand 47, 323, 325
 - change* built-in program 323
 - Example of use 47, 50, 69
 - Channel command word 150
 - Glossary definition 952
 - Channel commands 150
 - char* syntax variable 224
 - buildscr* 319
 - polish* 527
 - tape* 634
 - CHARS program option 345
 - Cheque protection 188
 - chop* built-in program 326, 133
 - Example of use 6, 12, 19, 36, 56, 58, 59, 60, 62, 64, 68, 132
 - cipher* built-in program 328
 - ckddebloc* built-in program 330
 - CLEAR
 - REXX 115
 - CLOSE 79, 543
 - CLOSE 151
 - cmd* built-in program 339, 337, 337
 - CMS
 - Glossary definition 952
 - cms* built-in program 331
 - Example of use 18, 41
 - CMSRET 506
 - CNCTD
 - REXX 16
 - CNCTDN
 - REXX 12
 - CNTLOG
 - EXEC 13
 - Code point 677
 - collate* built-in program 332, 87
 - Collating sequence 228
 - Glossary definition 952

Index

- combine* built-in program 335
 - command* built-in program 339, 337
 - Example of use 41, 42, 70, 127
 - Command environments 114
 - Comments
 - Formatting 26
 - COMMIT 246
 - COMMIT pipeline command 754, 246
 - Commit level 247
 - Compatibility 933
 - Computational-3 171
 - Computed output position 181
 - Configuration variable
 - Diskreplace 273
 - Disktempfiletype 273
 - Configuration variables 867
 - Conditional operator 192
 - configure* built-in program 340, 867
 - Configuring *CMS Pipelines* 867
 - Connection 2
 - Example of use 11, 12, 16, 18, 19
 - Glossary definition 952
 - Connector
 - Example of use 64
 - Glossary definition 952
 - Connectors 241
 - Console
 - Glossary definition 952
 - Path 414
 - console* built-in program 341
 - Example of use 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 20, 33, 34, 35, 44, 52, 77, 124
 - CONSOLE Query 418
 - Console queue 341
 - Console stack 34
 - Consumer stage 767, 249
 - Consumes 249
 - Consuming read 249
 - Control break 195
 - Control stage
 - Glossary definition 952
 - Controls 253
 - Conversion 175, 571, 711
 - Zoned decimal 711
 - COPY 56
 - REXX 98, 763
 - copy* built-in program 343
 - COPYFILE 918
 - COPYFILE 50
 - COPYFILE Option
 - EBCDIC 918
 - FILL 918
 - FOR 918
 - FRLABEL 51, 918
 - FROM 918
 - LOWCASE 918
 - COPYFILE Option (*continued*)
 - LRECL 918
 - OLDDATE 918
 - OVLY 918
 - PACK 918
 - RECFM 918
 - SINGLE 918
 - SPECS 918
 - TOLABEL 51, 918
 - TRANS 918
 - TRUNC 918
 - UNPACK 918
 - UPCASE 918
 - COPYND
 - REXX 99, 762
 - coroutines 113
 - count* built-in program 344
 - Example of use 3, 5, 6, 11, 13, 15, 46, 63, 636
 - Counter 178
 - Counters 722
 - countlms* built-in program 808
 - CP
 - Glossary definition 952
 - cp* built-in program 345
 - Example of use 8, 13, 14, 17, 40, 79
 - CP Monitor 589
 - CP System Services
 - *ACCOUNT 594, 595
 - *LOGREC 594, 595
 - *MSG 592
 - *MSGALL 592
 - *SYMPTOM 594, 595
 - cpasis* built-in program 808
 - CPHRASAMP
 - EXEC 330
 - CPRC
 - REXX 768
 - CRC 347
 - crc* built-in program 347
 - Cycle 720, 166
 - Cyclic Redundancy Code 347
- ## D
- dam* built-in program 351
 - DAMSAMP
 - REXX 352
 - DANAID
 - EXEC 524
 - Data stream
 - Glossary definition 952
 - dateconvert* built-in program 352
 - DateTimeSubtract callable service 833
 - DB2 583
 - DDNAME 539
 - SYSTSPRT 119

- ddname* syntax variable 234
 - <*mvs* 265
 - >*mvs* 274
- Defined 79
- deal* built-in program 360, 424
- deblock* built-in program 365, 133, 267
 - Example of use 62, 65, 66
- Decimal Sorting 128
- Declaring a label 242
- DECODE 930
- Decoding trees 82
- Default configuration variable 870
- delay* built-in program 369
- Delay the record 89
- delimitedString* syntax variable 224
 - <*sfs* 267
 - <*sfs*slow 268
 - >>*sfs* 283
 - >>*sfs*slow 285
 - >*mvs* 274
 - >*sfs* 276
 - affst* 293
 - all* 295
 - beat* 308
 - between* 309
 - block* 310
 - buffer* 317
 - change* 323
 - chop* 326
 - cipher* 328
 - console* 341
 - deblock* 365
 - escape* 387
 - fbawrite* 394
 - fntfst* 404
 - ftp* 409
 - insert* 450
 - inside* 450
 - join* 458
 - joincont* 460
 - locate* 472
 - nlocate* 501
 - notinside* 505
 - outside* 508
 - pick* 518
 - readpds* 544
 - rexxvars* 549
 - sfsback* 558
 - sfsdirectory* 559
 - sfsrandom* 561
 - sfsupdate* 563
 - space* 569
 - spec* 571, 729
 - spill* 577
 - split* 580
 - starsys* 595
- delimitedString* syntax variable (continued)
 - strasmfind* 609
 - strasmnfind* 610
 - strfind* 611
 - strfrlabel* 612
 - strip* 613
 - strliteral* 614
 - strnfind* 616
 - strtolabel* 617
 - strwhilelabel* 625
 - tcpclient* 638
 - tcpdata* 643
 - timestamp* 652
 - tokenise* 654
 - trackwrite* 661
 - varfetch* 683
 - varload* 685
 - varset* 688
 - vchar* 690
 - verify* 692
 - wildcard* 701
 - writepds* 703
- delimiter* syntax variable 224
 - change* 323
- delover* built-in program 512
- Destructive selection 84
- devaddr* syntax variable 224
 - browse* 315
 - devinfo* 371
 - diage4* 373
 - fbaread* 393
 - fbawrite* 394
 - fullscr* 413
 - fullscrq* 418
 - fullscrs* 419
 - mapmdisk* 485
 - printmc* 534
 - punch* 536
 - reader* 542
 - trackread* 659
 - trackwrite* 661
 - uro* 674
 - waitdev* 697
 - xab* 705
- Device driver 2
 - Glossary definition 952
 - Isolating from main pipeline 90
- Device drivers 253, 7
- devinfo* built-in program 371
- dfsrt* built-in program 372
- DFSORT/CMS 372
- DFSRTLIB TXTLIB 372
- diage4* built-in program 373
- Diagnose
 - 14 542
 - 24 418

Index

- Diagnose (*continued*)
 - 8C 321, 418
 - A8 536, 537, 676
 - Diagnose A8 824
 - Diagnose instruction
 - Glossary definition 952
 - digest* built-in program 374
 - digit* syntax variable 224
 - <*sfs* 267
 - <*sfs**slow* 268
 - >>*sfs* 283
 - >>*sfs**slow* 285
 - >*sfs* 276
 - sfsback* 558
 - sfsrandom* 561
 - sfsupdate* 563
 - spec* 741
 - sysout* 631
 - DIRBUFF 623
 - DIRBUFF 560
 - Direct read 341
 - Direct read from terminal 341
 - dirid* syntax variable 231
 - <*sfs* 267
 - <*sfs**slow* 268
 - >>*sfs* 283
 - >>*sfs**slow* 285
 - >*sfs* 276
 - sfsback* 558
 - sfsrandom* 561
 - sfsupdate* 563
 - state* 597
 - statew* 601
 - Discard operator 184
 - diskback* built-in program 376
 - diskfast* built-in program 376
 - diskid* built-in program 378
 - DISKID 378, 857
 - diskrandom* built-in program 378
 - Diskreplace 273
 - Diskreplace configuration variable 868
 - diskslow* built-in program 379
 - Disktempfiletype 273
 - Disktempfiletype configuration variable 868
 - diskupdate* built-in program 380
 - Dispatcher
 - Glossary definition 952
 - DISPLAY
 - XEDIT Subcommand 708
 - DMSDTS 833
 - DMSEXIST 599, 603
 - DMSFILEC 868
 - DMSGETDI 560, 623, 836
 - DMSGETWU 268, 269, 278, 284, 287, 558, 561, 564
 - DMSOPBLK 823
 - DMSOPDBK 823
 - DMSPIPE 867
 - DMSPQI
 - ASMSQL 585
 - dmsstoar* built-in program
 - DMSVALDT 599, 603
 - DONE program option 725
 - DOS
 - Glossary definition 953
 - Dotted-decimal IP address 638
 - Dotted-decimal network address 638
 - Drift 185
 - Drifting sign 747, 186
 - Driver
 - Glossary definition 953
 - drop* built-in program 381
 - Example of use 56, 79, 82
 - DSN (DB2 Subsystem ID) 583
 - dsname* syntax variable 234
 - <*mvs* 265
 - >*mvs* 274
 - listispf* 468
 - listpds* 469
 - readpds* 544
 - writepds* 703
 - Dual speed carriage 150
 - DUMLOAD 874
 - duplicate* built-in program 382
- ## E
- E
 - Picture character 748
 - EBADF 817
 - EBCDIC
 - COPYFILE Option 918
 - EDF
 - Glossary definition 953
 - elastic* built-in program 384
 - ELSIF program option 725
 - emsg* built-in program 385
 - ENAMETOOLONG 818
 - end character 74, 19
 - Example of use 19, 77, 81
 - Glossary definition 953
 - End-of-file 90
 - Backwards propagation 90
 - endChar* syntax variable 224
 - ENDCHAR pipeline option 238
 - ENDIF program option 725
 - Enhanced Disk Format
 - Glossary definition 953
 - ENOENT 817, 818
 - ENOTDIR 817
 - Entry point table 927

- Enumerate
 - Glossary definition 953
 - Enumerated scalar 953
 - EOF program option 183
 - block* 310
 - console* 341
 - deblock* 365
 - eofback* built-in program 386
 - EOFREPORT pipeline command 755
 - ERASE 41
 - escape* built-in program 387
 - ESCAPE pipeline option 238
 - Escape character 120
 - Glossary definition 953
 - European pictures 749
 - EVENTRECORD 623
 - EVERY
 - REXX 161
 - Exactness latch 722
 - EXEC 342
 - 3800 35
 - CNTLOG 13
 - CPHRSAMP 330
 - DANAID 524
 - GENMLIB 80
 - Glossary definition 953
 - HELLO2 548
 - HONK2 440
 - INSPIPE 456
 - NS0 129
 - NS1 130
 - PD 27
 - QCPSETS 686
 - REPRINT 79
 - RESBYU 84
 - SAMPLOC 77
 - SAYBAR 23
 - SPECNT 345
 - SQLINS 140
 - SQLQ3 141
 - SUBCOM 627
 - TESTALT 548
 - WHOCALLS 551
 - XMASTREE 516
 - EXEC2
 - Glossary definition 953
 - EXECCOMM 118, 551
 - EXECIO 919
 - EXECUPDT 25
 - Execution profiler 208
 - Extended attribute buffer 151
 - External function 116, 117
 - EXTRACT
 - XEDIT Subcommand 679
 - extract* built-in program 498
-
- F**
 - F
 - Glossary definition 953
 - F program option 228
 - fanin* built-in program 387
 - Example of use 78, 81, 87
 - faninany* built-in program 388
 - Example of use 77, 132
 - fanintwo* built-in program 389
 - fanout* built-in program 391, 90
 - fanoutwo* built-in program 392, 408, 657
 - Example of use 90
 - FB
 - Glossary definition 953
 - fbaread* built-in program 393
 - fbawrite* built-in program 394
 - fblock* built-in program 395, 133
 - FI program option 725
 - Field test version xx
 - FIELDS program option 228
 - FIELDSEPARATOR program option 168, 228
 - structure* 621
 - File
 - Glossary definition 953
 - File Status Table 293, 404
 - Glossary definition 953
 - fileback* built-in program 376
 - filedescriptor* built-in program 396
 - filefast* built-in program 377
 - FILELIST 27, 42
 - filerandom* built-in program 379
 - fileslow* built-in program 380
 - filetoken* built-in program 397
 - fileupdate* built-in program 381
 - FILL
 - COPYFILE Option 918
 - fillup* built-in program 399, 408, 657
 - Filter
 - Glossary definition 953
 - Filter package
 - Glossary definition 953
 - filterpack* built-in program 400
 - Filters 253
 - FIND
 - XEDIT Subcommand 44
 - find* built-in program 402
 - Example of use 14, 42, 53, 66, 67, 68
 - FINDANY
 - REXX 532
 - First stage
 - Glossary definition 953
 - fitting* built-in program 404
 - fltpack* built-in program 924
 - Flush
 - Glossary definition 953

Index

- fm* syntax variable 231
 - <mnsk* 264
 - >>mnsk* 279
 - listpds* 469
 - mnskback* 490
 - mnskfast* 488
 - mnskrandom* 491
 - mnskslow* 493
 - mnskupdate* 495
 - members* 496
 - pdsdirect* 517
 - rexx* 547
 - state* 597
 - statew* 601
 - xedit* 705
- fmode* syntax variable 231
 - >mnsk* 271
- fmfst* built-in program 404
- FMTF
 - XEDIT 24, 121
- FMTPCBIN
 - REXX 368
- fn* syntax variable 231
 - <mnsk* 264
 - <sfs* 267
 - <sfsslow* 268
 - >>mnsk* 279
 - >>sfs* 283
 - >>sfsslow* 285
 - >mnsk* 271
 - >sfs* 276
 - listpds* 469
 - mnskback* 490
 - mnskfast* 488
 - mnskrandom* 491
 - mnskslow* 493
 - mnskupdate* 495
 - members* 496
 - pdsdirect* 517
 - rexx* 547
 - sfsback* 558
 - sfsrandom* 561
 - sfsupdate* 563
 - state* 597
 - statew* 601
 - xedit* 705
- FOR
 - COPYFILE Option 918
- FORCERW program option 785
- Formatting a pipeline 24
- Formatting comments 26
- Forms control buffer 150
 - Glossary definition 953
- FPL global variable group 867
- FPLASIT 291, 623
- FPLEPTBL 927
- FPLGREXX 930
- FPLHELP 23
- FPLHLASX 439
- FPLKWDTB 929
- FPLMSGTB 928
- FPLNXG
 - TEXT 926
- FPLNXH
 - TEXT 926
- FPLMOM
 - MACLIB 623
- FPLREXX 23, 97
- FPLSTORBUF 623
- FRLABEL
 - COPYFILE Option 51, 918
- fritel* built-in program 406
 - Example of use 55, 56
- FROM
 - COPYFILE Option 918
- fromlabel* built-in program 407
- fromtarget* built-in program 408
- frtarget* built-in program 407
- FS program option 168, 228
- FST
 - See File Status Table
- ft* syntax variable 231
 - <mnsk* 264
 - <sfs* 267
 - <sfsslow* 268
 - >>mnsk* 279
 - >>sfs* 283
 - >>sfsslow* 285
 - >mnsk* 271
 - >sfs* 276
 - listpds* 469
 - mnskback* 490
 - mnskfast* 488
 - mnskrandom* 491
 - mnskslow* 493
 - mnskupdate* 495
 - members* 496
 - pdsdirect* 517
 - rexx* 547
 - sfsback* 558
 - sfsrandom* 561
 - sfsupdate* 563
 - state* 597
 - statew* 601
 - xedit* 705
- ftp* built-in program 408
- Full block interface 78
- fullscr* built-in program 413
- fullscrq* built-in program 418
- fullscrs* built-in program 419

Function pool variables 145
 Functional programming xix

G

G

Glossary definition 953
gate built-in program 422
 Gateway
 Glossary definition 953
 Gateways 253
gather built-in program 423
generation syntax variable 234
 <*mvs* 265
 >>*mvs* 281
 >*mvs* 274
 listispf 468
 listpds 469
 members 496
 readpds 544
 state 599
 writepds 703
 GENMLIB
 EXEC 80
 GENMOD 926
getfiles built-in program 425
 Example of use 5, 38, 39
 GETRANGE pipeline command 756
 Global option 19
 Global options 237
 GLOBAL TXTLIB 372
 GLOBALV 867
 Glue
 Glossary definition 953
greg2sec built-in program 426
 GRIDIT
 REXX 629
 Group configuration variable 869

H

HCPSGIOP 536, 537, 676, 796
 HEADING
 REXX 761
 HELLO
 REXX 97, 760
 HELLO2
 EXEC 548
 Help 23
help built-in program 427
hex syntax variable 224
 adrspace 290
 filedescriptor 396
 filetoken 397
 instore 451
 outstore 509

hex syntax variable (*continued*)

polish 527
 runpipe 553
 starmon 589
 storage 607
hexString syntax variable 224
 buildscr 319
 crc 347
 diskid 378
 mapmdisk 484, 485
 storage 607
 stsi 626
 tape 634
hfs built-in program 430, 267, 282
hfsdirectory built-in program 431
hfsquery built-in program 432
hfsreplace built-in program 433, 275
hfsstate built-in program 434
hfsxecute built-in program 435
 HLASM 813
hlasm built-in program 437
hlasmerr built-in program 439
hole built-in program 440
 Example of use 440
 HONK2
 EXEC 440
 Host
 Glossary definition 953
 Host command processors 253
 Host commands from REXX filters 114
 Host interface 2
 Glossary definition 953
 Host-primary address space 210
hostbyaddr built-in program 441
hostbyname built-in program 442
hostid built-in program 443
hostname built-in program 444
httpsplit built-in program 445
 HX 194

I

IBM 407 Accounting Machine 719
identifier syntax variable 225
 polish 526, 527
 spec 726, 734, 739, 742, 743, 746
 structure 618, 620, 621
 IEANTRT 835
 IEBCOPY
 Glossary definition 953
iebcopy built-in program 446
 IEWBFDAT 846
if built-in program 447
 IF program option 725
 IFEND program option 725

Index

- Ignored 719
 - Ignoring case 128
 - Ignoring case in comparisons 57
 - IKJCT441 552, 605, 680, 684, 687, 689
 - IKJEFTSR 821
 - immcmd* built-in program 448
 - Implied REXX Filters 117
 - INCLUDE 926
 - Infrastructure
 - Glossary definition 953
 - INITIAL GLOBALV 867
 - Input stream 2
 - Input translate table 708
 - inputRange* syntax variable 228, 50
 - change* 323
 - collate* 332
 - dateconvert* 352
 - deal* 361
 - gather* 423
 - insert* 450
 - ispf* 455
 - joincont* 460
 - lookup* 475
 - merge* 498
 - pick* 518
 - sort* 567
 - spec* 571, 729
 - substring* 628
 - threeway* 651
 - unique* 668
 - verify* 692
 - wildcard* 701
 - xlate* 708
 - zone* 714
 - inputRanges* syntax variable 225
 - locate* 472
 - nlocate* 501
 - insert* built-in program 450
 - inside* built-in program 450
 - Example of use 56
 - INSPIPE
 - EXEC 456
 - instore* built-in program 451
 - Interactive System Productivity Facility 145
 - Internal representation 175
 - Interpret REXX Instruction 100
 - Intersecting pipelines 76
 - Intersection
 - Glossary definition 953
 - Invisible input 342
 - Invocation 2
 - IP address 638
 - ip2socka* built-in program 454
 - IPaddress* syntax variable 225
 - ftp* 409
 - tcpclient* 638
 - IPaddress* syntax variable (*continued*)
 - tcplisten* 648
 - udp* 665
 - IPDS 417
 - IRXSTK 937
 - Isolating a device driver from main pipeline 90
 - ISPCMDS 148
 - ISPF 145
 - ispf* built-in program 455
 - ISSUEMSG pipeline command 757
 - Issuing 719
 - IUCV 589, 592, 595, 797
 - *MONITOR 589
- ## J
- jeremy* built-in program 457
 - JES 36
 - join* built-in program 458
 - Example of use 14, 17, 60
 - joincont* built-in program 460
 - Journeyman plumber 127
 - JRCompNameTooLong 818
 - JREndingSlashOCreat 818
 - JRPathTooLong 818
 - JRQuiescing 818
 - JRxxx reason codes 837
 - juxtapose* built-in program 462
 - Example of use 84
- ## K
- K
 - Glossary definition 953
 - Keyboards 23
 - Keyword table 929
- ## L
- Label 242, 79
 - Declaring 242
 - Glossary definition 954
 - Label reference 242
 - Landscape 24
 - Glossary definition 954
 - Landscape format 11
 - Language diskette 23
 - LAST program option
 - combine* 336
 - drop* 381
 - Example of use 54, 55
 - take* 633
 - unique* 668
 - update* 672
 - Last stage
 - Glossary definition 954

- ldrtbls* built-in program 464
- LEADING program option
 - Example of use 59, 67, 68
 - joincont* 460
 - strip* 613
- Left
 - Glossary definition 954
- Left-handed pipeline 75
- Length of records 131
- letter* syntax variable 226
 - mdiskblk* 487
 - spec* 724, 728, 729, 739, 742, 743, 746
 - structure* 620
 - sysout* 631
- Libraries 80
- Limit on REXX clause 243
- Line
 - Glossary definition 954
- Line end character
 - Glossary definition 954
- Line splicing 61
- LINK
 - Address 935
- LIST3820 369
- listcat* built-in program 465
 - Example of use 72
- LISTCMD pipeline option 238
- listdsi* built-in program 466
 - Example of use 73
- LISTERR pipeline option 238
- LISTFILE 5, 41, 70
- LISTFILE 702
- listispf* built-in program 468
- listpds* built-in program 469
 - Example of use 6
- LISTRC pipeline option 238
- LITAFTER
 - REXX 104
- literal* built-in program 471
 - Example of use 30, 34, 44, 56
- LOAD 926
- LOCAL program option 427, 557
 - starsys* 595
- Local options 237, 122
- locate*
 - OR of targets 128
- locate* built-in program 472
 - Example of use 52, 67, 68, 77
- LOCATE 819
- locate mode read 249
- Look up routines 253
- lookup* built-in program 474, 87
 - Example of use 88
- Loss of precision 722
- LOWCASE
 - COPYFILE Option 918
- LRECL
 - COPYFILE Option 918
- M**
- M
 - Glossary definition 954
- Machine carriage control character 150
- MACLIB
 - FPLM 623
- maclib* built-in program 483
 - Example of use 81, 82
- MACLIB 483
- Macro libraries 80
- MACWRITE
 - REXX 81
- MAKEDC
 - REXX 307
- mapmdisk* built-in program 484
- MAPMDISK 484
- MAPPDS
 - REXX 517
- Master file update 74
- MAXSTREAM 103
- MAXSTREAM pipeline command 758, 103
- mctoasa* built-in program 486
- mdiskblk* built-in program 487
- mdskback* built-in program 490
- mdskfast* built-in program 488
- mdskrandom* built-in program 491
- mdskslow* built-in program 493
- mdskupdate* built-in program 495
- member* syntax variable 234
 - <*mvs* 265
 - >*mvs* 274
 - readpds* 544
- members* built-in program 496, 545
- merge* built-in program 498, 87
- MESSAGE pipeline command 759
- Message level
 - Glossary definition 954
- Message text table 928
- Mixed case file names 41, 231
- Monitor System Service 589
- MONWRITE 590, 591
- move mode read 249
- MOVEFILE 919
- mjsc* built-in program 499
- MRHDR 590
- MSGLEVEL pipeline option 238
- Multi Lingual Plane 676
- Multilevel update 86
- Multistream pipelines 2, 18, 74, 101
- MVS
 - Glossary definition 954

Index

MVS SPOOL 36
MVULTAPE
 REXX 636
MYASIT
 REXX 212
MYFILTER
 REXX 931

N

NAME pipeline option 238
NAME_MAX 818
NAMEDEF 264, 267, 268, 271, 276, 279, 282, 285, 377, 380,
 557, 560, 562
NAMESAVE 591
NETDATA 920
 Glossary definition 954
NETDATA program option
 block 310
 deblock 365
 Example of use 65
Network address 638
NEXT program option 172
 spec 571
 structure 620
NEXTW program option 168
NEXTWORD program option 168
 spec 571
nfind built-in program 500
 Example of use 15, 67, 68
ninside built-in program 505
nlocate built-in program 501
 Example of use 10, 11, 16
NOCOMMIT pipeline command 759
noeofback built-in program 503
not built-in program 503
NOT program option
 chop 326
 digest 374
 Example of use 59
 joincont 460
 spill 577
 split 580
 strip 613
Not connected 74
noteofback built-in program 503
notfind built-in program 617, 501
notinside built-in program 505
notlocate built-in program 503
NOTREADY 796
NS0
 EXEC 129
NS1
 EXEC 130
nucext built-in program 506

Null
 Glossary definition 954
Null fields. 168
Null record
 Glossary definition 954
Null stage
 Glossary definition 954
NUMBER program option 173
 Example of use 49
 filetoken 397
 mdiskblk 487
 mdskrandom 491
 sfsrandom 561
 spec 571
 vchar 690
number syntax variable 226
 <sfs 267
 <sfsslow 268
 >>mdsk 279
 >>sfs 283
 >>sfsslow 285
 >mdsk 271
 >sfs 276
 3277bfra 715
 abbrev 287
 asmcont 303
 beat 308
 between 309
 block 310
 browse 315
 buffer 317
 buildscr 319
 combine 336
 cp 345
 dateconvert 352
 deal 361
 deblock 365
 devinfo 371
 diage4 373
 drop 381
 fanout 391
 fbaread 393
 fbawrite 394
 fblock 395
 filedescriptor 396
 ftp 409
 gather 423
 greg2sec 426
 help 428
 hfsxecute 436
 inside 450
 join 458
 lookup 475
 mapdisk 484, 485
 mdskfast 488
 mdskslow 493

number syntax variable (continued)

mdskupdate 495
notinside 505
outside 508
pack 512
pad 514
pick 518
pipcmd 523
random 541
reader 542
retab 545
rexxvars 549
runpipe 553
scm 555
sec2greg 556
sfsback 558
sfsrandom 561
sfsupdate 563
snake 565
space 569
spec 571, 724, 726, 734, 737, 739, 746
spill 577
split 580
stem 603
storage 607
strip 613
structure 620
take 633
tape 634
tcpcksum 637
tcpclient 638
tcpdata 643
tcplisten 648
timestamp 652
trackread 659
trackwrite 661
trfread 663
udp 665
untab 671
var 678
vardrop 681
varfetch 683
varload 685
varset 688
xab 705
xlate 708
xpndhi 713
Number representation 711
 Zoned decimal 711
Numeric Sorting 128
numorstar syntax variable 226
 change 323
NW program option 168

O

octalDigit syntax variable 226
 hfsxecute 437
OLDDATE
 COPYFILE Option 918
OMVS 349
OpenExtensions Files 31
Operating System
 Glossary definition 954
optcdj built-in program 507
Options
 See Pipeline option
 See Program option
 See Syntax variable
OR for *locate* targets 128
OS record descriptor words 62
OTHERWISE program option 725
OUTDES 632
OUTPUT pipeline command 760
OUTPUT 632
Output stream 2
Output translate table 708
outside built-in program 508
outstore built-in program 509
OUTSTREAM program option 178
OUTTRAP 41
overlay built-in program 510
Overlong encodings 677
Overriding a built-in program 117
overstr built-in program 511
OVLY
 COPYFILE Option 918

P

PACK
 COPYFILE Option 918
pack built-in program 512
Packed numbers 722
pad built-in program 514
 Example of use 17, 66, 81
PAD program option 174, 175
 >>*mvs* 281
 >>*sfs* 283
 >>*sfs*slow 285
 >*mvs* 274
 >*sfs* 276
 collate 332
 filetoken 397
 lookup 475
 merge 498
 pick 518
 sort 567
 spec 571
 unique 668

Index

- PAD program option (*continued*)
 - vchar* 690
 - writepds* 703
- Pad character 174
- parcel* built-in program 515
- PARSERANGE 765
- PARSERANGE pipeline command 765
- PARSESTRING 766
- PARSESTRING pipeline command 766
- Parsing error 182
- Past data 83
- Path
 - console 414
- PATH_MAX 818
- pause* built-in program 516
- PD
 - EXEC 27
- pdsdirect* built-in program 517, 471
- pdslist* built-in program 471
- pdslisti* built-in program 469
- pdsread* built-in program 545
- pdswrite* built-in program 704
- PEEKTO pipeline command 761
- PFX
 - REXX 100
- PGM= 342
- pick* built-in program 518
 - Example of use 69
- Picture 747, 185, 178
 - Default 748
 - European conventions 749
- pipcmd* built-in program 523
- PIPDATE
 - REXX 71
- PIPDESC 245
- PIPDUMP 126
- PIPE 863, 864
- Pipeline
 - Formatting 24
 - Glossary definition 954
- Pipeline command
 - Glossary definition 954
- Pipeline configuration variables 867, 867
- pipeline dispatcher 2
 - Glossary definition 954
- Pipeline option 238—239
 - Defined 120
- Pipeline specification 2
- Pipeline specification
 - Glossary definition 954
- Pipeline stall 88
- pipestop* built-in program 525
- Pipethink 7
- PIPGFMOD 924
- PIPGFTEXT 924
- PIPMOD 863
- PIPMOD INSTALL 932
- PIPNXF
 - TEXT 926
- PIPPTFF filter package 117
- PIPSCBLK 940
- PIPSCSTG 942
- PIPSQI 801
- PIPWECB 246
- PIPXSAMP
 - XEDIT 244
- Placement option 735, 173
- Plan (DB2) 583
- pods* syntax variable 235
 - listispf* 468
 - readpds* 544
 - writepds* 703
- polish* built-in program 525
- Portrait 24
 - Glossary definition 954
- Portrait format 12
- Potential to delay a record 251
- Prefix connector 241
- Precision, loss of 722
- predselect* built-in program 531
 - Example of use 85
- preface* built-in program 532
- Preparing the access module 138
- Primary data stream
 - Glossary definition 954
- Primary stream 76, 74
- Pring Services Facility 151
- PRINT 920
- Print Services Facility 350, 508, 512
- Printer 151
 - Glossary definition 954
- Printer Carriage Control 152
- Printing a counter 179
- printmc* built-in program 534, 632
 - Example of use 35, 36, 79
- Producer stage 767, 249
- PROFILE WTPMSG 23, 119
- Profiler 208
- Program Access key 1 415
- Program option
 - : 409
 - (295
 -) 295
 - * 319, 336, 381, 382, 458, 541, 591, 595, 597, 601, 631, 633, 672, 701
 - % 701
 - 0 319
 - 00C 542
 - 00E 705
 - 1 298, 299, 319, 336, 381, 382, 458, 633, 634
 - 15 310, 365

Program option (*continued*)

16-BIT 347
 2 298, 299, 319
 32-BIT 347
 3277 298, 299, 319
 3278 298, 299, 319
 3279 298, 299, 319
 3DES 328
 3F 310, 365
 4KBLOCK 542
 8 652
 80 326, 365
 8192 345
 A 409
 ACCT 410
 ADD 296, 618
 ADDLENGTH 347
 ADMSF 310, 365
 AES 328
 AFTER 326, 450, 518, 580
 ALET 290, 451, 509, 607
 ALL 400, 465
 ALLEOF 361, 391, 423, 523, 571
 ALLMASTER 475
 ALLOWEMPTY 276, 283, 285, 563
 AND 336, 518
 ANY 638, 648, 665
 ANYCASE 287, 309, 323, 326, 332, 450, 460, 472, 475,
 498, 501, 505, 508, 518, 567, 577, 580, 609, 610, 611,
 612, 613, 616, 617, 618, 625, 668, 692, 701
 ANYCHARACTER 701
 ANYEOF 332, 361, 391, 423, 523, 571
 ANYOF 326, 460, 472, 501, 569, 577, 580, 613
 ANYSTRING 701
 APL 298, 299, 319
 APPEND 347, 374, 409, 603, 614
 ASA 631
 ASCENDING 498, 567
 ASCII 409
 ASIS 267, 268, 276, 283, 285, 465, 544, 558, 561, 563,
 597, 601
 ASSEMBLER 525
 ASYNCHRONOUS 595
 ASYNCHRONOUSLY 341, 413, 665, 667
 AT 580
 AUTOADD 475
 AUTOFIELD 621
 AUTOSTOP 389
 AWTAPE 310, 365
 BACKLOG 648
 BACKWARDS 397
 BASEYEAR 352
 BEFORE 326, 450, 475, 580
 BLANK 510, 514, 580, 613, 701
 BLOCKED 397, 491, 561
 BLOWFISH 328

Program option (*continued*)

BOTH 613
 BROADCAST 665
 BUILTIN 618
 BUILTINS 428
 BY 571
 BYTES 381, 633
 C 310, 365
 CALLER 618
 CASEI 714
 CBC 328
 CC 315
 CCITT-16 347
 CDATE 276
 CEILING 475
 CENTRE 571
 CHARACTERS 344
 CHDIR 436
 CHMOD 436
 CHOP 274, 276, 281, 283, 285, 397, 703
 CKSUM 347
 CLASS 631
 CMS 289, 310, 365
 CMS4 289, 310, 365
 CODEPAGE 708
 COERCE 274, 276, 281, 283, 285, 397, 703
 COMMANDS 428
 COMMENTS 549, 683, 685, 688
 COMMIT 582
 COMPLEMENT 347
 CONDITIONAL 614
 CONDREAD 413
 CONNECT 582
 COUNT 458, 462, 475, 518, 567, 668
 CP 663
 CRC-16 347
 CRC-16I 347
 CRC-32 347
 CRCFIRST 347
 CREATE 290
 CRLF 310, 365, 638, 643
 CSL 353
 D 409, 527
 DARK 341
 DATACODEPAGE 315
 DB2 353
 DDNAME 265, 274, 281, 468, 469, 496, 544, 599, 703
 DEBLOCK 638, 643
 DECIMAL 365
 DECRYPT 328
 DEFAULT 267, 268, 276, 283, 285, 558, 561, 563
 DEFINE 484
 DELAY 460
 DELETE 618
 DELIMITER 544, 703
 DES 328

Index

Program option (*continued*)

DESCENDING 498, 567
DESCRIBE 582
DESTINATION 631
DESTROY 290
DETAIL 332, 475
DETAILS 409
DIRECT 341, 603, 678, 681, 683, 685, 688
DROP 400
DSNAME 465
EACH 347
EBCDIC 409, 445, 673
EMSGSF4 638
ENCRYPT 328
EOF 310, 341, 365
EOTOK 634
ESM 267, 268, 276, 283, 285, 558, 561, 563
EUR 353
EVENTS 553, 589
EXCLUSIVE 365, 589, 612, 617
EXCLUSIVEOR 336
EXECUTE 582
FETCH 484
FIELD 621
FIELDSEPARATOR 621
FIFO 588
FILE 542, 705
FIRST 336, 381, 633, 668, 672
FIXED 271, 276, 279, 283, 285, 310, 365, 488, 493, 495,
512, 563
FLOOR 475
FORMAT 559, 597, 601
FROM 268, 285, 493, 518, 571, 603, 676, 708
FROM16BIT 715
FROMRIGHT 365
FTP:// 409
FTPS:// 409
FULLDATE 293, 353, 404, 559, 597, 601, 652
FULLPACK 373
FULLVOL 373
GDFORDERS 365
GETSECINFO 638
GETSOCKNAME 638, 643, 648, 665
GLOBAL 400
GREETING 638, 643
GROUP 638, 643
HARDEN 285, 563
HEADING 400
HEXADECIMAL 525
HOLD 542
HOST 428
HOSTID 638, 648, 665
I 409, 527
IDENTIFIED BY 582
IDENTIFY 484
IFEMPTY 614

Program option (*continued*)

IMMEDIATE 391
INCLUSIVE 365, 612, 617
INCREMENT 475
INDELIMITER 703
INDICATORS 582
INITIALISE 290
INPLACE 276
INPUT 708
INSERT 582
INTERNAL 448
INTO 582
IPUSER 595
ISODATE 293, 353, 404, 559, 597, 601, 652
ISOLATE 290
ISPFSTATS 274, 703
IV 328
JULIAN 353
K 527
KEEP 276, 283, 285, 460, 542, 563, 577
KEEPALIVE 638, 643
KEY 361
KEYLENGTH 336, 458
KEYONLY 475
L 527
LAST 336, 381, 633, 668, 672
LATCH 361
LEADING 460, 613
LEFT 514, 571
LENGTH 620
LEVEL 540
LIFO 588
LINEND 310, 365, 638, 643
LINES 344
LINGER 638, 643
LINK 436
LIST 400, 618
LISTALL 618
LOAD 400
LOCAL 595
LOCALIPADDRESS 638, 648, 665
LOCALPORT 638
LOCALTIME 663
LOWER 708
MACHINE 631
MAIN 549, 603, 678, 681, 683, 685, 688
MASK 553
MASTER 332, 475
MAXCOUNT 475
MAXLINE 344
MD5 374
MDATE 276, 283, 285, 563
MEMBER 620
MEMBERS 544, 618
MENU 428
MESSAGES 428

Program option (*continued*)

MET 353
 MINIMUM 580
 MINLINE 344
 MIXED 472, 501
 MKDIR 436
 MKFIFO 436
 MODIFIED 676
 MODLIST 400
 MODULO 514
 MONITOR 365
 MONWRITE 589
 MSG 428
 MSGLEVEL 540, 553
 MSGLIST 540
 MULTIPLE 668
 N 527
 NETDATA 310, 365
 NEXT 571, 620
 NEXTWORD 571
 NOBSCAN 455
 NOCHOP 276, 283, 285, 397
 NOCLOSE 413
 NOCOMMENTS 549, 683, 685, 688
 NOCOMMIT 582
 NODETAILS 597, 601
 NOEOF 341
 NOFORMAT 293, 434, 559, 597, 601
 NOHOLD 542
 NOINDICATORS 582
 NOKEEP 542
 NOMSG233 549, 603, 678, 681, 683, 685, 688
 NOPAD 276, 283, 285, 332, 397, 475, 498, 518, 567, 668
 NOREAD 413
 NORECOVER 276
 NORMAL 353
 NOSPIN 631
 NOT 326, 374, 460, 577, 580, 613
 NUMBER 397, 487, 491, 561, 571, 690
 O 527
 OFF 701
 OFFSET 303, 426, 556, 577
 OLDDATeref 267, 268, 558, 561
 ONCE 308
 ONEBYTE 365
 ONERESPONSE 638, 643
 ONES 472, 501
 OOBINLINE 638, 643
 OPENRECOVER 268, 285, 558, 561, 563
 OR 336, 518
 ORDER 455
 OSPDSDir 446
 OTHER 428
 OUTDESCRIPTOR 631
 OUTPUT 708
 PAD 274, 276, 281, 283, 285, 332, 397, 475, 498, 518,
 567, 571, 668, 690, 703

Program option (*continued*)

PADIN 690
 PADOUT 690
 PAIRWISE 475, 668
 PARMLIST 595
 PASS 410
 PATH 413
 PERMIT 290
 PGMLIST 451
 PGMOWNER 582
 PIPE 353
 PLAN 582
 POSIX 353
 PREFACE 614
 PRELOAD 347
 PRIVATE 267, 268, 276, 283, 285, 558, 561, 563
 PRMDATA 595
 PRODUCER 549, 603, 678, 681, 683, 685, 688
 PURGE 542
 QLINK 373
 QMDISK 373
 QUERY 290
 QUIET 434, 597, 599, 601
 RANDOM 397
 RANGE 460
 RDW 365
 READ 487, 571, 607
 READER 705
 READFULL 413
 READSTOP 571
 RECEIVE 696
 RECNO 571
 RECURSIVE 559
 REFLIN 347
 REFLOUT 347
 RELEASE 361, 582
 REMOVE 296, 484
 RENAME 436
 REPEATABLE 582
 REPORT 676
 RESOLVE 400
 RETAIN 484
 REUSEADDR 638, 643, 648, 665
 REVERSE 322, 451, 714
 RFC959 365
 RIGHT 514, 571
 RMDIR 436
 ROWCOL 715
 S 527
 SAFE 276, 283, 285, 409, 563, 638
 SAMPLES 589
 SAVE 484
 SCIABS 353
 SCIREL 353
 SECONDARY 361
 SECURE 638, 643

Index

Program option (*continued*)

SELECT 571, 582
SET 618
SETCOUNT 475
SF 289, 310, 365, 638, 643
SF4 289, 310, 365, 638, 643
SHA1 374
SHA224 374
SHA256 374
SHA384 374
SHA512 374
SHARED 589, 703
SHORTDATE 293, 353, 404, 559, 597, 601, 652
SHR 274
SINGLES 668
SITE 409
SPECS 723
SPIN 631
SQL 428
SQLCODE 428
SQUISH 499
STANDARD 293, 404, 559, 597, 601, 652
STATISTICS 638, 643, 648, 665
STOP 332, 361, 391, 423, 523, 534, 536, 571, 674
STREAMID 361, 423
STRICT 388, 422, 475
STRING 293, 310, 326, 365, 394, 404, 559, 569, 577,
580, 613, 638, 643, 652, 661
STRIP 361, 365, 571
STRIPKEY 446
STRUCTURE 620
SUBSYSID 582
SUPPRESS 589
SYMBOLIC 603, 678, 681, 683, 685, 688
SYMLINK 436
SYNTAX 428
T 527
TAP 634
TARGET 595
TBADD 455
TBMOD 455
TBPUT 455
TBSKIP 455
TCPIP 409
TERMCODEPAGE 315
TERMINATE 310, 365, 458, 658
TEST 296
TEXT 298, 299, 319
TEXTFILE 310, 365
TEXTUNIT 365
THREAD 400, 618
TIMEOUT 352, 638
TSLABEL 643
TO 518, 582, 613, 676, 708
TO16BIT 715
TOD 328

Program option (*continued*)

TODABS 353
TODCLOCK 571
TODREL 353
TOLOAD 549, 683
TOROWCOL 715
TRACE 409, 553
TRACKCOUNT 475
TRACKING 678
TRAILING 460, 613
TRSOURCE 663
TRUNCATE 690
TYPE 409
UDP 667
UMASK 436
UNIQUE 409, 567
UNLINK 436
UNSAFE 409, 638
UPDATE 397
UPPER 708
USA 353
USER 290, 410
USERDATA 274, 544, 703
USERID 443, 444, 638, 648, 665
UTF-16 676
UTF-32 676
UTF-8 676
UTIME 436
VARIABLE 271, 276, 279, 283, 285, 289, 310, 365, 488,
493, 495, 512, 563
VB 310
VBS 310
VCIT 290
VCOPY 455
VERIFY 374
VERSION 540
VMDATE 353
VREPLACE 455
VS 310
WAIT 413
WHILE 518
WIDTH 549, 715
WINDOW 352
WORD 621
WORDS 344
WORDSEPARATOR 621
WORKUNIT 267, 268, 276, 283, 285, 558, 561, 563
WRITE 290, 296, 487, 571
WTM 634
X 319
XOROUT 347
ZERO 484
ZEROS 472, 501
ZONE 322
Program stack 341

Programmable Operator 163
 PROP 163
 PROPAGATE pipeline option 238
 PRTPAGE
 REXX 206
psds syntax variable 235
 <*mvs* 265
 PSF 151, 350, 508, 512
 PTF filter package 924
 Public
 Glossary definition 954
 PUNCH 920
 Glossary definition 954
punch built-in program 536, 632
 Example of use 36, 40, 64
 Punched card 149
 Glossary definition 955

Q

QCPSETS
 EXEC 686
qpdecode built-in program 537
qpencode built-in program 538
qsam built-in program 539
 Qualifier 180
qualifier syntax variable 226
 QUALIFY pipeline option 238
query built-in program 540
 QUERY NAMES 7
 Quietly
 Glossary definition 955
quotedString syntax variable 226
 <*oe* 266
 >>*oe* 282
 >*oe* 275
 spec 739

R

random built-in program 541
 RANGE
 XEDIT Subcommand 708
range syntax variable 226
 change 323
 deblock 365
 filetoken 397
 ispf 455
 mapmdisk 485
 mdiskblk 487
 mdskrandom 491
 sfsrandom 561
 spec 571, 734
 structure 621
 update 672

RDROP
 REXX 769
 Re-entrant REXX environments on MVS 119
 READ program option 176
 mdiskblk 487
 spec 571
 storage 607
 READCARD 920
reader built-in program 542
 Example of use 35, 36, 65, 79
readpds built-in program 544, 498
 Example of use 6
 READSTOP program option 176
 spec 571
 READTO pipeline command 762
 READY 795, 796
 REALUSER
 REXX 15
 RECFM
 COPYFILE Option 918
 Record
 Glossary definition 955
 Record delay 250, 89
 Record descriptor word
 Glossary definition 955
 Record descriptor words 62
 RECORDS program option 345
 Redefine connector 241
 reentrant environments 115
 Referenced 79
 Referencing a label 242
 REFRESH
 XEDIT Subcommand 712
 Register 179
 Remembering past data 83
 Repairing LIST3820 369
 Repository configuration variable 869
 REPRINT
 EXEC 79
 RESBYU
 EXEC 84
 RESOLVE pipeline command 763
 Resume 246
retab built-in program 545
 Return codes
 -102 770
 -111 755
 -112 752, 755, 758, 759, 763, 767, 768, 770, 771
 -113 755, 757, 767
 -122 785
 -163 758, 768, 770, 771
 -164 758, 768, 770, 771
 -168 767
 -169 767
 -174 752
 -178 770

Index

Return codes (*continued*)

-2147483648 754
-3 116
-4 767, 770, 771
-4095 252, 760, 761, 762
-42 763
-58 757, 767
-60 757
-7 116, 252, 751
-9 252
0 752, 753, 754, 755, 757, 758, 759, 760, 761, 762,
763, 767, 768, 770, 771
12 98, 760, 761, 762, 767, 771
13 272, 281, 490, 494, 496
16 272, 281, 490, 494, 496
20 265, 272, 281, 457, 490, 491, 492, 494, 496
24 265, 273, 281, 490, 491, 493, 494, 496
25 265, 273, 281, 490, 491, 493, 494, 496
4 759, 760, 761, 767, 771
8 755, 759, 761, 767, 771

reverse built-in program 546

REWORD1

REXX 90

REXX 97

ALLDIRS 471
ASYNCMS 163
BOTHDISK 670
C2X 212
CATTWO 388
CLEAR 115
CNCTD 16
CNCTDN 12
COPY 98, 763
COPYND 99, 762
CPRC 768
DAMSAMP 352
EVERY 161
FINDANY 532
FMTPCBIN 368
Glossary definition 955
GRIDIT 629
HEADING 761
HELLO 97, 760
LITAFTER 104
MACWRITE 81
MAKEDC 307
MAPPDS 517
MVULTAPE 636
MYASIT 212
MYFILTER 931
PFX 100
PIPDAT 71
PRTPAGE 206
RDRDP 769
Re-entrant environments 119
REALUSER 15

REXX (*continued*)

Reentrant environments 115
REWORD1 90
RXP 100
RXPDP 113
RXPI 101
SCOMP 102
TAGNSPL 154
TCALLP 754
TCMT 760
TCOMMT 755
THROTTLE 363
TISSUE 758
TMAXSTR 758
TMSG 759
TPOS 769
TRES 763
TREXXC 764
TSTRNO 770
TSTRST 771
USER2NAM 18
USERTERM 19
VMCSERV 158
ZONE2DEC 711

rexx built-in program 546

REXX clause

Limit on size 243

REXX Compiler Runtime Environment 547

REXXCMD pipeline command 763

rexxvars built-in program 549

Right

Glossary definition 955

Rita 208

RSCS 163

RSTD 591

Run list 246

Runin 195

Runin cycle 720

Runout 195

Runout cycle 720, 183

runpipe built-in program 553

Runtime library xx

Runtime profiler 208

RXP

REXX 100

RXPDP

REXX 113

RXPI

REXX 101

RXSOCKET 225, 442, 443

S

S

Picture character 747

- SAMPLOC
 - EXEC 77
- SAYBAR
 - EXEC 23
- Scanning rules 956
- SCANRANGE pipeline command 764
- SCANSTRING pipeline command 766
- SCBLOCK 782
- SCBLOCK 506
- SCIF 164
- SCM
 - XEDIT 26, 556
- scm* built-in program 555
- SCOMP
 - REXX 102
- Scope
 - Glossary definition 955
- sec2greg* built-in program 556
- Second reading station 720, 194
- Secondary data stream
 - Glossary definition 955
- Secondary stream 76
- Segment descriptor word
 - Glossary definition 955
- SEGMENT LOAD 589
- Segmentation Flags
 - Glossary definition 955
- SELECT program option 197
 - spec* 571
 - sql* 582
- SELECT pipeline command 766
- Selecting records 82
- Selection by irreversible modification 84
- Selection stages 253
- Semantics 222
- SEPARATOR pipeline option 238
- Sequential data stream
 - Glossary definition 955
- Service programs 253
- SET EMSG ON 773
- SET MSGLINE
 - XEDIT Subcommand 712
- SET PREFIX 31
- SET SPILL WORD 579
- SETRC pipeline command 767
- Sever 617, 655
 - Glossary definition 955
- SEVER pipeline command 768
- SFBLOK 151
- sfsback* built-in program 557
- sfsdirectory* built-in program 559
- sfsrandom* built-in program 560
- sfsupdate* built-in program 562
- SGIOP 536, 537, 676
- Shared File System
 - Glossary definition 955
- Shell
 - Glossary definition 955
- SHORT pipeline command 769
- Short circuit 34, 104, 617, 655
 - Glossary definition 955
- Short-through connection 242, 239
- SHVBLOCK 793
- SHVNEWV 684, 689
- Sign 722
- SINGLE
 - COPYFILE Option 918
- Single Console Image Facility 164
- Skip to a channel 150
- snake* built-in program 565
- snumber* syntax variable 227
 - chop* 326
 - dateconvert* 352
 - duplicate* 382
 - random* 541
 - spec* 571, 729, 731, 746
 - split* 580
 - structure* 620
- socka2ip* built-in program 566
- Solicited read 414
- sort* built-in program 567
 - Example of use 17, 37, 62, 63, 70
 - Problems with 133
- Sorters 253
- Sorting
 - Numeric 128
- Source entry point table 927
- Source keyword table 929
- Source message text table 928
- Source virtual machine 156
- space* built-in program 569
- SPACE program option 227
- Span
 - Glossary definition 955
- spec* built-in program 719, 571, 133
 - Example of use 13, 18, 19, 35, 40, 42, 49, 50, 58, 62, 65, 66, 68, 69, 71, 79
- spec* parsing error 182
- specification items 166
- Specification list 719, 571
- SPECNT
 - EXEC 345
- SPECS
 - COPYFILE Option 918
- spill* built-in program 577
- Spilling words 577
- Splicing lines 61
- split* built-in program 580, 133
 - Example of use 9, 10, 11, 16, 56, 58, 59, 60, 63
- Splitting records 133
- SPOOL
 - Glossary definition 955

Index

- sql* built-in program 582
 - Example of use 139
- SQL/DS 138
- sqlcodes* built-in program 587
- SQLDBSU 586
- SQLINS
 - EXEC 140
- SQLpgmname configuration variable 869
- SQLpgmowner configuration variable 869
- SQLQ3
 - EXEC 141
- sqlselect* built-in program 587
 - Example of use 20, 139
- Stack 341
- stack* built-in program 588
 - Example of use 440
- Stage 8, 2
 - Glossary definition 955
- Stage separator 23
 - Glossary definition 955
- Stage separator character 23
- STAGENUM pipeline command 769
- stageSep* syntax variable 227
- STAGESEP pipeline option 238
- Stall 88
 - Glossary definition 955
- Stallaction configuration variable 870
- Stalled 88
- Stallfiletype configuration variable 870
- starmon* built-in program 589
- starmsg* built-in program 591
 - Example of use 44
- starsys* built-in program 594
- state* built-in program 599, 597
 - Example of use 70, 71, 72
- statew* built-in program 600
- STATS program option 640, 645, 648, 665
- STAX 816
- stderr* built-in program 397
- stdin* built-in program 397
- stdout* built-in program 397
- stem* built-in program 603
 - Example of use 36, 37, 127, 636
- stembuild* built-in program 480
- Stemmed array 36
 - Glossary definition 955
- Stepwise refinement 7
- stfle* built-in program 606
- STOP pipeline option 238
- STOPERROR pipeline option 239
- Stopping an infinite pipeline 161
- storage* built-in program 607
- STOW 829
- strasmfind* built-in program 609
- strasmnfind* built-in program 610
- Stream
 - Glossary definition 955
- stream* syntax variable 227
 - fanin* 387
 - rex* 547
 - spec* 571, 727
- stream identifier 103
- Stream identifier
 - Glossary definition 955
- streamID* syntax variable 227
- STREAMNUM pipeline command 770
- STREAMSTATE pipeline command 770
- strfind* built-in program 611
 - Example of use 65
- strfrlabel* built-in program 612
- strfromlabel* built-in program 613
- STRING program option
 - affst* 293
 - block* 310
 - chop* 326
 - deblock* 365
 - Example of use 58, 59
 - fbawrite* 394
 - fintfst* 404
 - sfsdirectory* 559
 - space* 569
 - spill* 577
 - split* 580
 - strip* 613
 - tcpclient* 638
 - tcpdata* 643
 - timestamp* 652
 - trackwrite* 661
- string* syntax variable 227
 - < 263
 - > 270
 - >> 278
 - append* 300
 - asmfind* 304
 - asmnfind* 306
 - casei* 322
 - change* 323
 - cms* 331
 - command* 337, 339
 - cp* 345
 - dfsrt* 372
 - diskback* 376
 - diskfast* 377
 - diskrandom* 378
 - diskslow* 379
 - diskupdate* 380
 - eofback* 386
 - find* 402
 - frrlabel* 406
 - frtarget* 407
 - help* 428

string syntax variable (continued)

- hfs* 430
- hfsdirectory* 431
- hfsquery* 432
- hfsreplace* 433
- hfsexecute* 437
- hiasm* 437
- if* 447
- ldrtbls* 465
- listdsi* 466
- literal* 471
- nfind* 500
- not* 503
- nucext* 506
- preface* 532
- rexx* 547
- sql* 582
- starmsg* 591
- starsys* 595
- subcom* 626
- sysdsn* 630
- tolabel* 655
- totarget* 656
- tso* 664
- update* 672
- vmc* 693
- whilelabel* 700
- zone* 714
- strip* built-in program 613
 - Example of use 9, 10, 11, 16, 19, 58, 59, 67, 68
- STRIP program option 184
 - deal* 361
 - deblock* 365
 - spec* 571
- strliteral* built-in program 614
- strnfind* built-in program 616
 - Example of use 65, 69
- strtolabel* built-in program 617
- structure* built-in program 618
- Structured Fields 417
 - 3270DS 417
- strwhilelabel* built-in program 625
- stsi* built-in program 626
- Style configuration variable 870
- SUBCOM
 - EXEC 627
- subcom* built-in program 626
 - Example of use 42
- Subcommand Environment
 - Glossary definition 956
- Subroutine pipeline 103, 10, 2, 11, 103
 - Glossary definition 956
- subscript* syntax variable 227
 - structure* 620
- substring* built-in program 628

- Suppress leading zeros 747
- Surrogate pair 677
- SUSPEND pipeline command 772
- SVC
 - Glossary definition 956
- SVC 202
 - Glossary definition 956
- SVC 203
 - Glossary definition 956
- synchronise* built-in program 628
- Synchronises 251, 177
- synchronize* built-in program 628
- syncsort* built-in program 372
- Syntax 222
- Syntax variable 224—236
- sysdsn* built-in program 630
- SYSIN 36
- SYSOUT 36
- sysout* built-in program 631
- SYSTSPRT 23
- SYSTSPRT DDNAME 119
- sysvar* built-in program 632

T

- TABULATE program option 227
- Tag 151
- TAGNSPL
 - REXX 154
- take* built-in program 633
 - Example of use 9, 54, 55, 66
- tape* built-in program 634
 - Example of use 62
- TAPn 635
- Target virtual machine 156
- Task
 - Glossary definition 956
- TCALLP
 - REXX 754
- TCMT
 - REXX 760
- TCOMMT
 - REXX 755
- TCP/IP 665
- tcpcksum* built-in program 637
- tcpclient* built-in program 638
- tcpdata* built-in program 643
- tcplisten* built-in program 648
- Terminal 342
- terminal* built-in program 341, 341
- Terminal Monitor Program
 - Glossary definition 956
- Terminals 23
- Terminate prematurely 254
- Terminology xxi
- Tertiary data stream
 - Glossary definition 956

Index

- TESTALT
 - EXEC 548
 - Testing by destruction 84
 - TEXT
 - ARIRVSTC 800, 801
 - FPLNXG 926
 - FPLNXH 926
 - PIPNXF 926
 - Text file 31
 - threeway* built-in program 651
 - THROTTLE
 - REXX 363
 - timestamp* built-in program 652
 - Timestamp string sample 653
 - TISSUE
 - REXX 758
 - TMAXSTR
 - REXX 758
 - TMSG
 - REXX 759
 - TO program option
 - Example of use 67, 68
 - pick* 518
 - sql* 582
 - strip* 613
 - utf* 676
 - xlate* 708
 - Token
 - Glossary definition 956
 - Tokenise
 - Glossary definition 956
 - tokenise* built-in program 654
 - tokenize* built-in program 654, 654
 - TOLABEL
 - COPYFILE Option 51, 918
 - tolabel* built-in program 655
 - Example of use 44, 55, 56
 - Topology diagram 19
 - totarget* built-in program 656
 - TPOS
 - REXX 769
 - TRACE pipeline option 239
 - trackblock* built-in program 657
 - trackdeblock* built-in program 658
 - trackexpand* built-in program 663
 - trackread* built-in program 659
 - tracksquish* built-in program 660
 - trackverify* built-in program 660
 - trackwrite* built-in program 661
 - trackxpan* built-in program 662
 - TRAILING program option
 - Example of use 59
 - joincont* 460
 - strip* 613
 - TRANS
 - COPYFILE Option 918
 - translate* built-in program 708, 708
 - Transparent
 - Glossary definition 956
 - TRES
 - REXX 763
 - TREXXC
 - REXX 764
 - trfread* built-in program 663
 - TRUNC
 - COPYFILE Option 918
 - truncate* built-in program 326, 326
 - TSO
 - Address 935
 - Glossary definition 956
 - tso* built-in program 664
 - TSO Logon Procedure 23
 - TSTRNO
 - REXX 770
 - TSTRST
 - REXX 771
 - TXTLIB 372
 - DFSRTLIB 372
 - TYPE 920
 - Type 1 filter package 924
 - Type 2 filter package 924
- ## U
- udp* built-in program 665
 - Unconnected pipeline specification 248
 - unique* built-in program 668
 - Example of use 56, 63
 - Unit record
 - Glossary definition 956
 - Unlimited
 - Glossary definition 956
 - UNPACK
 - COPYFILE Option 918
 - unpack* built-in program 670
 - Example of use 35, 56, 62, 69, 80
 - untab* built-in program 671
 - UPCASE
 - COPYFILE Option 918
 - UPDATE 920
 - update* built-in program 672
 - Example of use 87
 - UPDATE 86
 - UPPER program option
 - Example of use 44, 62, 63
 - xlate* 708
 - Upper case translation 708
 - urldeblock* built-in program 673
 - uro* built-in program 674
 - USER2NAM
 - REXX 18

USERTERM

REXX 19

utf built-in program 676**V**

V

Glossary definition 956

Picture character 748

var built-in program 678*vardrop* built-in program 681*varfetch* built-in program 683

VARIABLE program option

>>*mdsk* 279>>*sfs* 283>>*sfs**slow* 285>*mdsk* 271>*sfs* 276*addrdw* 289*block* 310*deblock* 365

Example of use 62

mdskfast 488*mdskslow* 493*mdskupdate* 495*pack* 512*sfsupdate* 563

Variable length record format 62

varload built-in program 685*varset* built-in program 688

VB

Glossary definition 956

VBS

Glossary definition 956

VBS program option

block 310

Example of use 124

vchar built-in program 690

Verb

Glossary definition 956

verify built-in program 692

Virtual configuration identification token 211

Virtual unit record 149

vmc built-in program 693*vmcdata* built-in program 694

VMCF address space 825

vmclient built-in program 695*vmclisten* built-in program 696

VMCMPARM 623

VMCSERV

REXX 158

VMDUMP 542

VMDUMP 873

VMDUMPTL 874

VMMHDR 623

*vm**sort* built-in program 372**W***w* program option 167, 228*waitdev* built-in program 697

Warm start data 150

warp built-in program 698*warplist* built-in program 699

WCC 415

WHILE program option 725

pick 518*whilelabel* built-in program 700

WHOCALLS

EXEC 551

wildcard built-in program 701*word* syntax variable 227<*oe* 266>>*mvs* 281>>*oe* 282>*oe* 275*abbrev* 287*adr**space* 290*casei* 322*diage4* 373*eofback* 386*fbawrite* 394*filterpack* 400*fitting* 404*frtarget* 407*ftp* 409, 410*fullscr* 413*help* 428*hfs* 430*hfsdirectory* 431*hfsquery* 432*hfsreplace* 433*hfsxecute* 437*hiasm* 437*hostid* 443*hostname* 444*if* 447*immcmd* 448*is**pf* 455*ldrtbls* 465*listcat* 465*listis**pf* 468*listpds* 469*maclib* 483*members* 496*m**qsc* 499*not* 503*nucext* 506*optcdj* 507*qsam* 539*readpds* 544

Index

word syntax variable (continued)

- scm* 555
- sfsdirectory* 559
- spec* 729, 741
- sql* 582
- starmon* 589
- starmsg* 591
- starsys* 595
- state* 599
- stem* 603
- subcom* 626
- sysout* 631
- sysvar* 632
- tcpclient* 638
- tcpdata* 643
- tcplisten* 648
- totarget* 656
- trackwrite* 661
- udp* 665
- var* 678
- vmc* 693
- vmclient* 695
- warp* 698
- writepds* 703
- zone* 714

Word spill 577

WORDS program option 167, 228

- count* 344

WORDSEPARATOR program option 228

- structure* 621

WRITE program option 177

- adrspc* 290
- alserv* 296
- mdiskblk* 487
- spec* 571

Write Control Character 415

writepds built-in program 703

Writing REXX filters 3, 97

WS program option 228

WTPMSG 23, 119

X

XAB 151

xab built-in program 705

XEDIT

- FMTP 24, 121
- Glossary definition 956
- PIPXAMP 244
- SCM 26, 556

xedit built-in program 705

- Example of use 65

XEDIT Macro

- Glossary definition 956

XEDIT Subcommand

- ALL 44

XEDIT Subcommand (continued)

- CHANGE 47, 323, 325

- DISPLAY 708

- EXTRACT 679

- FIND 44

- RANGE 708

- REFRESH 712

- SET MSGLINE 712

- ZONE 47

xeditmsg built-in program 712, 712

xithlp03 built-in program 429

xlake built-in program 708

- Example of use 44, 45, 46, 62, 63

XMASTREE

- EXEC 516

XMIT 36

XMITMSG 109

xmsg built-in program 712

xorc syntax variable 227

- >>*mvs* 281

- >>*sfs* 283

- >>*sfs*slow 285

- >*mvs* 274

- >*sfs* 276

- block* 310

- c14to38* 350

- collate* 332

- deblock* 365

- fblock* 395

- filetoken* 397

- lookup* 475

- merge* 498

- overlay* 510

- pad* 514

- pick* 518

- sort* 567

- space* 569

- spec* 571, 726

- storage* 607

- structure* 621

- tcpclient* 638

- tcpdata* 643

- unique* 668

- vchar* 690

- wildcard* 701

- writepds* 703

- xrange* 713

xpndhi built-in program 713

xrange built-in program 713

xrange syntax variable 228

- chop* 326

- split* 580

- strip* 613

- xlake* 708

- xrange* 713

xtract built-in program 498, 808

Y

Y

Picture character 748

Z

Z

Picture character 748

z/OS format V 62

ZONE

XEDIT Subcommand 47

zone built-in program 714

Example of use 67

ZONE2DEC

REXX 711

Zoned decimal 711



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC24-6252-01

