

Pipelines for NetRexx QuickStart Guide

Ed Tomlinson

Jeff Hennick

René Jansen

Version 3.08-GA of September 8, 2019

THE REXX LANGUAGE ASSOCIATION
NetRexx Programming Series
ISBN 978-90-819090-3-7

Publication Data

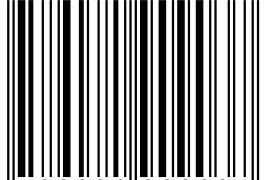
©Copyright The Rexx Language Association, 2011- 2019

All original material in this publication is published under the Creative Commons - Share Alike 3.0 License as stated at <http://creativecommons.org/licenses/by-nc-sa/3.0/us/legalcode>.

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

This edition is registered under ISBN 978-90-819090-3-7

ISBN 978-90-819090-3-7



9 789081 909037 >

Contents

The NetREXX Programming Series	i
Typographical conventions	iii
1 Introduction	1
2 The Pipeline Concept	2
2.1 What is a Pipeline?	2
2.2 Stage	2
2.3 Device Driver	3
3 Running pipelines	4
3.1 Configuration	4
3.2 From the NetREXX Workspace (nrws) with direct execution	5
3.3 From the command line with direct execution	5
3.4 Precompiled Pipelines	5
3.5 Compiled from an .njp file	6
3.6 Compiled from an .njp file with additional stage definitions in NetREXX	6
4 Example Session	8
5 Write your own Filters	12
6 More advanced Pipelines	14
7 Device Drivers	15
8 Record Selection	16
9 Filters	17
10 Other Stages	18
11 Multi-Stream Pipelines	19
12 Pipeline Stalls	21

13 Differences with CMS Pipelines	23
14 How to use a pipe in a NetREXX program	24
15 TCP/IP Networking using Pipes for NetREXX	27
16 Selecting from databases with Pipelines for NetREXX	29
17 The Pipes Runner	30
18 The Pipes Compiler	31
19 Built-in Stages	32
20 Appendix A	46
List of Figures	54
List of Tables	54
Index	57

The NetREXX Programming Series

This book is part of a library, the *NetREXX Programming Series*, documenting the NetREXX programming language and its use and applications. This section lists the other publications in this series, and their roles. These books can be ordered in convenient hardcopy and electronic formats from the Rexx Language Association.

Quick Start Guide	This guide is meant for an audience that has done some programming and wants to start quickly. It starts with a quick tour of the language, and a section on installing the NetREXX translator and how to run it. It also contains help for troubleshooting if anything in the installation does not work as designed, and states current limits and restrictions of the open source reference implementation.
Programming Guide	The Programming Guide is the one manual that at the same time teaches programming, shows lots of examples as they occur in the real world, and explains about the internals of the translator and how to interface with it.
Language Reference	Referred to as the NRL, this is the formal definition for the language, documenting its syntax and semantics, and prescribing minimal functionality for language implementors. It is the definitive answer to any question on the language, and as such, is subject to approval of the NetREXX Architecture Review Board on any release of the language (including its NRL).
Pipelines for NetREXX QuickStart Guide	The Data Flow oriented companion to NetREXX, with its z/VM CMS Pipelines compatible syntax, is documented in this manual. It discusses installing and running Pipes for NetREXX, and has ample examples of defining your own stages in NetREXX.

Typographical conventions

In general, the following conventions have been observed in the NetRexx publications:

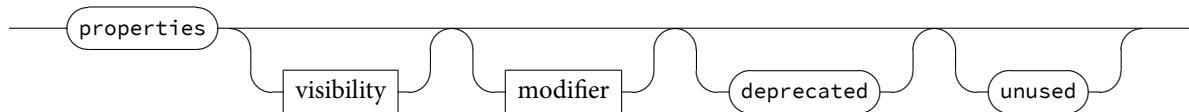
- Body text is in this font
- Examples of language statements are in a **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are listed in this fashion:

Listing 1: Example Listing

```
1 -- salute the reader  
2 say 'hello reader'
```

- Syntax diagrams take the form of so-called *Railroad Diagrams* to convey structure, mandatory and optional items

Properties



Introduction

A Pipeline, or Hartmann Pipeline¹², is a concept that extends and improves pipes as they are known from Unix and other operating systems. The name pipe indicates an inter-process communication mechanism, as well as the programming paradigm it has introduced. Compared to Unix pipes, Hartmann Pipelines offer multiple input- and output streams, more complex pipe topologies, and a lot more.

Pipelines were first implemented on VM/CMS, one of IBM's mainframe operating systems. This version was later adapted to run under MUSIC/SP and TSO/MVS (now z/OS) and has been part of several product configurations. Pipelines are widely used by VM users, in a symbiotic relationship with REXX, the interpreted language that also has its origins on this platform.

Pipes for NetREXX is the implementation of Pipelines for the Java Virtual machine. It is written in NetREXX and pipes and stages can be defined using this language. It can run on every platform that has a JVM (Java Virtual Machine) installed. This portable version of Pipelines was started by Ed Tomlinson in 1997 under the name of *njPipes*, when NetREXX was still very new, and was open sourced in 2011, soon after the NetREXX translator itself. The included stages have always been open source. It was integrated into the NetREXX translator in 2014 and first released with version 3.04.

In version 3.08, there are important improvements that enable pipelines to be run from the command line, and from the NetREXX REPL program *nrws*, the NetREXX Workspace. The pipes compiler has been renamed *pipc*, while the pipes runner component keeps using the name *pipe*.

¹//https://en.wikipedia.org/wiki/CMS_Pipelines

²This page used to be called Hartmann Pipeline, but was renamed to CMS Pipelines in 2016

The Pipeline Concept

2.1 What is a Pipeline?

The *pipeline* terminology is a set of metaphores derived from plumbing. Fitting two or more pipe segments together yields a pipeline. Water flows in one direction through the pipeline.

There is a source, which could be a well or a water tower; water is pumped through the pipe into the first segment, then through the other segments until it reaches a tap, and most of it will end up in the sink. A pipeline can be increased in length with more segments of pipe, and this illustrates the modular concept of the pipeline.

When we discuss pipelines in relation to computing we have the same basic structure, but instead of water that passes through the pipeline, data is passed through a series of programs (*stages*) that act as filters.

Data must come from some place and go to some place. Analogous to the well or the water tower there are *device drivers* that act as a source of the data, where the tap or the *sink* represents the place the data is going to, for example to some output device as your terminal window or a file on disk, or a network destination.

Just as water, data in a pipeline flows in one direction, by convention from the left to the right.

2.2 Stage

A program that runs in a pipeline is called a *stage*. A program can run in more than one place in a pipeline - these occurrences function independent of each other.

The pipeline specification is processed by the *pipeline compiler*, and it must be contained in a character string; on the commandline, it needs to be between quotes, while when contained in a file, it needs to be between the delimiters of a NetREXX string. An solid vertical bar | is used as *stage separator*, while other characters can be used as an option when specifying the local option for the pipe, after the pipe name.³

When looking a two adjacent segments in a pipeline, we call the left stage the *producer* and the stage on the right the *consumer*, with the *stage separator* as the connector.

³In versions before Pipelines for NetREXX 3.08, the default was the exclamation mark (!), which use was discontinued in favour of conformity with VM/CMS Pipelines.

2.3 Device Driver

A *device driver* reads from a device (for instance a file, the command prompt, a machine console or a network connection) or writes to a device; in some cases it can both read and write. An example of a device drivers are < and > ; these read and write data from and to files.

A pipeline can take data from one input device and write it to a different device. Within the pipeline, data can be modified in almost any way imaginable by the programmer.

The simplest process for the pipeline is to read data from the input side and copy it unmodified to the output side. Chapter 7 on page 15 shows the currently supported input- and output devices. The pipeline compiler connects these programs; it uses one program for each device and connects them together.

The inherent characteristic of the pipeline is that any program can be connected to any other program because each obtains data and sends data through a device independent standard interface. This becomes apparent when data can be in-line (specified or generated within the pipeline specification), come in (or be output) to devices like disk or tape, or be handled through a network – all these formats can be processed by the same stages.

The pipeline usually processes one record (or line) at a time. The pipeline reads a record for the input, processes it and sends it to the output. It continues until the input source is drained.

Running pipelines

There are a number of ways to specify and run a pipeline. A little setup is necessary.

3.1 Configuration

The required configuration is minimal. The NetREXXF.jar (java archive file) needs to be on the classpath environment variable (NetREXXC.jar, which is smaller, will suffice when there is a working javac compiler). Also, the current directory (.) needs to be on the classpath. It is convenient to have aliases or shell scripts defined as abbreviations for the invocation of the pipe, pipc (pipe compiler) and nrc (netrexx compiler) utility programs. Aliases are preferable because some shell processors have idiosyncrasies in the treatment of script arguments. With an alias we can be sure that every NetREXX program sees its arguments the same way.

```
.bash_aliases:  
alias pipc="java org.netrexx.njpipes.pipes.compiler"  
alias pipe="java org.netrexx.njpipes.pipes.runner"  
alias nrc="java org.netrexx.process.\nr{}C"
```

For Windows, the following works:

```
pipe.bat:  
@java org.netrexx.njpipes.pipes.runner %*
```

These aliases (or command script (in Windows it is called a batch file) enable you to do the following:

To run a pipeline from the commandline, type:

```
1 pipe 'gen 100 | dup 999 | count words | console'
```

Remember to use double quotes on Windows shells. When the pipe alias or command script is not on your path, you can also use:

```
1 java org.netrexx.njpipes.pipes.runner 'gen 100 | dup 999 | count words | console'
```

In both cases the answer should be 100000 - you have generated one hundred thousand lines, but fortunately you did not print them, but only counted them. To see them all, you can insert a | console | stage in between the dup and the count stage.

After we have verified the working of the command processors, we will discuss in the next section which possibilities you have for running pipelines in day-to-day usage.

3.2 From the NetREXX Workspace (nrws) with direct execution

The first way is the most straightforward, and highly recognizable for users of CMS Pipelines, as it mimics the way a pipe is run in the CMS 3270 interface. It also yields the best response time, specially when the `nrws.input` file in your home directory preloads the Pipes subsystem, as in this example:

```
-- preload the pipe machinery for good response on first pipe
pipe literal Pipelines processor loaded. | console
```

This is not magic: we do a Pipe execution (that displays: “Pipe processor loaded”) which loads all necessary classes and leaves them in memory. We can then type this command after the `nrws>` prompt.



A screenshot of a terminal window titled "java org.vpad.extra.workspace.Workspace". The window shows the output of a pipe command. It starts with the Java classpath and copyright information. Then it shows the command: "nrws> pipe literal a man a plan a canal panama | reverse | console". The output shows the reversed string "amanap lanac a nalp a nam a". Below the output, there are two timing metrics: "0.000 s" and "0.205 s". The prompt "nrws>" is visible at the bottom.

FIGURE 1: Run in the NetREXX Workspace

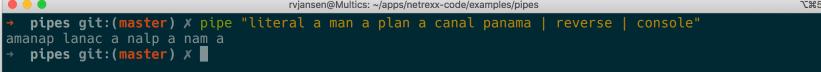
```
1 pipe literal a man a plan a canal panama | reverse | console
```

Executed this way, the executed class image will not be written to disk. The *timing* option is great for prototyping and performance work.

3.3 From the command line with direct execution

The only difference is that after the `pipe` command, the rest of the specification needs to be quoted in the command shells of Linux, Windows and macOS. In CMS, the pipeline specification can also be quoted - in this way, a pipeline can be entirely portable. Windows needs double quotes, zVM/CMS does not need quotes, but if they are used they need to be double quotes. macOS and Linux can use single or double quotes, in most cases.

```
1 pipe "literal a man a plan a canal panama | reverse | console"
```



A screenshot of a terminal window titled "rvjansen@Multics: ~/apps/netrexx-code/examples/pipes". The window shows the output of a pipe command. It starts with the user's name and host. Then it shows the command: "pipe "literal a man a plan a canal panama | reverse | console"". The output shows the reversed string "amanap lanac a nalp a nam a". The prompt "rvjansen@Multics: ~/" is visible at the bottom.

FIGURE 2: Run from the OS command line

Executed this way, the executed class image again will not be written to disk.

3.4 Precompiled Pipelines

In this mode, which uses the `pipc` command (for pipe compiler), a `.class` file will be persisted to disk. This class can be run as many times as needed, without the overhead of

compilation. This also would be the right mode for pipes that take different arguments when re-run. The pipe name needs to be specified, and will be the class name. When the class name exists, it will be overwritten.

```
1 pipc "(test1) literal a man a plan a canal panama | reverse | console"
```

```
rvjansen@Multics: ~/apps/netrexx-code/examples/pipes
→ pipes git:(master) ✘ pipc "(test1) literal a man a plan a canal panama | reverse | console"
( test1 ) literal a man a plan a canal panama | reverse | console
→ pipes git:(master) ✘ file test1.class
test1.class: compiled Java class data, version 52.0 (Java 1.8)
→ pipes git:(master) ✘ java test1
amanap lanac a nap a nam a
→ pipes git:(master) ✘ █
```

FIGURE 3: Precompile a Pipeline from the OS command line

This will yield a

`test1.class`

classfile, which can be executed by the java virtual machine.

The file `test1.class` can be run with the command⁴:

```
java test1
```

Be sure to leave out the `.class` suffix when invoking `java`.

3.5 Compiled from an `.njp` file

When compiled from a file, the pipe specification must not be quoted. Pipes can be specified in so-called /emphPortrait Mode, which is the standard for more complex pipelines as it is easier to read. An example is:

```
1 pipe (appendtest)
2
3   gen 100 |
4   append gen 50 |
5   rexx locate /0/ |
6   console
```

3.6 Compiled from an `.njp` file with additional stage definitions in NetREXX

An example (`length1.njp`) is:

```
1 pipe (lengthp) < output.lst | length1 | console
2
3 import org.netrexx.njpipes.pipes.
4 class length1 extends stage final
5 method run()
6   do
7     loop forever
8     line = rexx peekto()
9     l = line.length
10    output(l l.d2x line)
11    readto()
```

⁴or an appropriate shortcut in modern shells

```
12      end
13  catch StageError
14    rc = rc()
15  end
16  exit(rc*(rc<>12))
```

In this example, the name of the generated pipe is lengthp, while the name of the custom stage is length1. Be sure to invoke the right class, invoking length1 will have the JVM complain about a non-existing main method. This class (lengthp) will be generated by the command:

```
pipc length1
```

note that the .njp suffix is optional when invoking the pipes compiler. When run, it tries to read the contents of the file length.nrx and will put out its lines, prepended by the line length in decimal and hex - because that is what the (in NetREXX) specified homegrown stage does.

Example Session

Imagine you have landed a job as programmer in an accounting firm, and on your first day there is a question about backups; *the backup process takes too long*. There is an urgent need to identify the files that are produced on this day. You know how to this, of course, it is only some 20 lines of code; use the File() API, fill a collection class (you are thinking of an ArrayList already), or a TreeMap to sort the File object on last modified date already, call an instance of the Calender class, run a comparison - get that compiled and test it a bit - an hour or so would be sufficient. Of course, you need to install the Java compiler, because all machines have Java nowadays, but just not the compiler. But if you want to really impress people, you should type in a command line and be done with it. For this you can use NetREXX pipelines. Fortunately, you emailed the NetREXXFjar to yourself so you save it on the machine, and you're in business right away; you add it to the classpath. Your first pipeline command should just test the waters. For this chapter, we will use the

`nrws`

program. You send a command into the pipeline, and get its output:

```
| pipe command ls -laFTl | console
```

```
test git:(master) x nrws
Workspace For NetRexx 3.08 build 1,524-20190902-1623
Copyright (c) Martin Lafaixa 2000
Copyright (c) parts RexxLA 2019
Pipelines processor loaded.
nrws> pipe command ls -laFTl | console
total 17256
drwxr-xr-x 254 rvjansen staff 8128 Sep 2 17:09:12 2019 ./
drwxr-xr-x 183 rvjansen staff 5856 Sep 2 17:07:54 2019 ../
-rw-r--r--@ 1 rvjansen staff 6148 Nov 5 10:30:11 2018 .DS_Store
drwxr-xr-x 16 rvjansen staff 512 Sep 2 17:07:51 2019 .git/
-rw-r--r-- 1 rvjansen staff 9 Oct 9 14:28:25 2017 .gitignore
-rw-r--r-- 1 rvjansen staff 392 Oct 9 14:28:25 2017 AddFile.nrx
-rw-r--r-- 1 rvjansen staff 784 Oct 9 14:28:25 2017 BaseChange.nrx
-rw-r--r-- 1 rvjansen staff 206 Oct 9 14:28:25 2017 Calculator.nrx
-rw-r--r-- 1 rvjansen staff 223 Oct 9 14:28:25 2017 CalculatorTest.nrx
-rw-r--r--@ 1 rvjansen staff 1580 Dec 12 09:28:37 2017 ChangeFile1.nrx
-rw-r--r--@ 1 rvjansen staff 871 Dec 12 09:28:37 2017 ChangeFile2.nrx
-rw-r--r--@ 1 rvjansen staff 1772 Dec 12 09:28:37 2017 ChangeFile3.nrx
-rw-r--r-- 1 rvjansen staff 897 Oct 9 14:28:25 2017 ChangeReport.nrx
-rw-r--r-- 1 rvjansen staff 228 Oct 9 14:28:25 2017 ChangedFiles.nrx
-rw-r--r-- 1 rvjansen staff 1430 Jul 19 20:26:21 2018 CompilerVersion.java
-rw-r--r-- 1 rvjansen staff 1280 Oct 22 16:39:19 2018 DownloadFile.nrx
-rw-r--r-- 1 rvjansen staff 3136 Dec 15 10:11:36 2018 DownloadPDS.nrx
-rw-r--r-- 1 rvjansen staff 4647 Jul 20 17:13:39 2018 DynamicHelloWorld.java
-rw-r--r-- 1 rvjansen staff 570 Jul 22 00:20:31 2018 FC1CompilerVersion.nrx
-rw-r--r-- 1 rvjansen staff 575 Oct 9 14:28:25 2017 EbcdicTest.java
-rw-r--r-- 1 rvjansen staff 1354 Aug 7 09:14:04 2019 EnzoFX$1.class
-rw-r--r-- 1 rvjansen staff 2997 Aug 7 09:14:04 2019 EnzoFX.class
-rw-r--r-- 1 rvjansen staff 2432 Aug 7 07:04:21 2019 EnzoFX.java
-rw-r--r-- 1 rvjansen staff 1402 Aug 7 10:23:18 2019 EnzoLCD$timer_.class
-rw-r--r-- 1 rvjansen staff 3282 Aug 7 10:23:18 2019 EnzoLCD.class
-rw-r--r-- 1 rvjansen staff 2142 Aug 8 06:46:49 2019 EnzoLCD.nrx
-rw-r--r-- 1 rvjansen staff 601 Oct 9 14:28:25 2017 Expr.g4
-rw-r--r-- 1 rvjansen staff 1258 Oct 9 14:28:25 2017 ExprJoyRide.java
-rw-r--r-- 1 rvjansen staff 413 Jul 24 08:45:05 2019 FormatNumber.nrx
-1116 Dec 15 05:23:52 2014 FtpPDS.nrx
```

FIGURE 4: example 1

The `ls` command with the flags is the unix way to get a directory listing - for Windows

we would use *dir*. In this case, we send the output into the pipeline, but as the last stage (called a pipe 'sink') occurs immediately after that, every line will be echoed on the console. A number of lines like these will be displayed on the console, as in *example 1*.

You see straight away that the relevant info is not in the first columns, and not in consecutive columns; we want to know the date (whether it is today or not) and not the time. So we filter this out of every line with a *specs* stage, as in *example 2*.

```
1 pipe command ls -laFTl | specs 42-47 1 58-* 8 | console
```

```
nrws> pipe command ls -laFTl | rex Specs 42-47 1 58-* 8 | console
Sep 2 2019 ./
Sep 2 2019 ../
Nov 5 2018 .DS_Store
Sep 2 2019 .git/
Oct 9 2017 .gitignore
Oct 9 2017 AddFile.nrx
Oct 9 2017 BaseChange.nrx
Oct 9 2017 Calculator.nrx
Oct 9 2017 CalculatorTest.nrx
Dec 12 2017 ChangeFile1.nrx
Dec 12 2017 ChangeFile2.nrx
Dec 12 2017 ChangeFile3.nrx
Oct 9 2017 ChangeReport.nrx
Oct 9 2017 ChangedFiles.nrx
Jul 19 2018 CompilerVersion.java
Oct 22 2018 DownloadFile.nrx
Dec 15 2018 DownloadDS.nrx
Jul 20 2018 DynamicHelloWorld.java
Jul 22 2018 ECJCompilerVersion.nrx
Oct 9 2017 EbcidicTest.java
Aug 7 2019 EnzoFX$1.class
Aug 7 2019 EnzoFX.class
Aug 7 2019 EnzoFX.java
Aug 7 2019 EnzoLCD$timer_.class
Aug 7 2019 EnzoLCD.class
Aug 8 2019 EnzoLCD.nrx
Oct 9 2017 Expr.n4
Oct 9 2017 ExprJoyRide.java
Jul 24 2019 FormatNumber.nrx
Dec 15 2018 FtpPDS.nrx
Oct 9 2017 GetShal.nrx
Oct 9 2017 HelloWorldMainTopic.java
Oct 9 2017 HexPrint.nrx
Mar 28 2018 IMG_0725_2.jpg
Mar 28 2018 IMG_0735_2.jpg
```

FIGURE 5: example 2

We can easily sort this, with almost no programming:

```
1 pipe command ls -laFTl | specs 42-47 1 58-* 8 | sort | console
```

So what now comes out of the pipeline is sorted (see *example 3*). But this is a bit funny, we would like to see chronological order of course, so we switch around some columns with another *specs* stage:

```
1 pipe command ls -laFTl | specs 42-47 1 58-* 8 | specs 7-11 1 1-6 7 12-* 12 | sort | console
```

which is very near to what we want (see *example 4*). Only thing to do now is to filter on the date. We use the *locate* stage and hardcode the date for now. Let's say it is the 2nd of March, 2019:

```
1 pipe command ls -laFTl | specs 42-47 1 58-* 8 | specs 7-11 1 1-6
2 7 12-* 12 | locate /2019 Mar 2/ | sort | console
```

As *example 5* shows, on that day there were only two files produced. Also, because this is a short list now, you can see that Pipelines runs this pipe in 0.157 seconds, because we switched on the time option in *nrws*. Normally, you would specify your pipeline in a file and use *portrait mode*: *commandtest.njp*:

```
1 pipe (newfiles)
```

```

nrws> pipe command ls -laFTl | rex specs 42-47 1 58-* 8 | sort | console
java org.vpad.extra.workpad.Workspace
Apr  2 2019 readZip.nrx
Apr  5 2019 testFile.nrx
Apr  8 2019 listzip.nrx
Apr 10 2019 a.test
Apr 10 2019 b.test
Apr 10 2019 unzip.nrx
Apr 12 2018 testTextLine.nrx
Apr 24 2018 hello.c
Apr 24 2018 hello.s
Aug  5 2019 clg.class
Aug  5 2019 clg.nrx
Aug  7 2019 EnzoFX$1.class
Aug  7 2019 EnzoFX.class
Aug  7 2019 EnzoFX.java
Aug  7 2019 EnzoLCD$timer_.class
Aug  7 2019 EnzoLCD.class
Aug  7 2019 RSAnrx.class
Aug  7 2019 RSAnrx.nrx
Aug  7 2019 TreeViewSample.class
Aug  7 2019 TreeViewSample.java
Aug  7 2019 TreeViewSampleNRX.class
Aug  7 2019 TreeViewSampleNRX.nrx
Aug  7 2019 testLoop100mljoen.cpp
Aug  7 2019 testLoop100mljoenNRX.class
Aug  7 2019 testLoop100mljoenNRX.nrx
Aug  7 2019 testLoop100mljoen_JAVA.class
Aug  7 2019 testLoop100mljoen_JAVA.java
Aug  7 2019 testMainTopic$buttonAction.class
Aug  7 2019 testMainTopic.class
Aug  8 2019 EnzoLCD.nrx
Aug 13 2018 bigdog.nrx
Aug 13 2018 dog.nrx
Aug 13 2019 dyn.class
Aug 13 2019 dyn.nrx
Aug 14 2018 iban.nrx

```

FIGURE 6: example 3

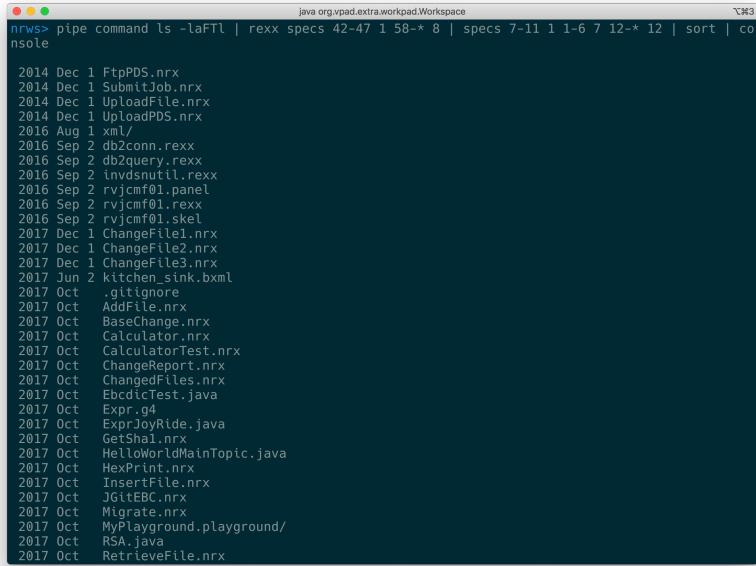
```

2 command ls -laFTl |
3 specs 42-47 1 58-* 8 |
4 specs 7-11 1 1-6 7 12-* 12 |
5 sort |
6 locate /2019 Mar 2/ |
7 console

```

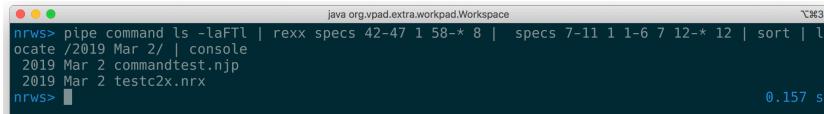
The filename is different from the generated class file name, on purpose. You could, and would, put different related pipelines in one file. Then we do a:

```
pipc commandtest && java newfiles
```



```
java org.vpad.extra.workpad.Workspace VM3
nrws> pipe command ls -laFTl | rexx specs 42-47 1 58-* 8 | specs 7-11 1 1-6 7 12-* 12 | sort | co
nsole
2014 Dec 1 FtpPDS.nrx
2014 Dec 1 SubmitJob.nrx
2014 Dec 1 UploadFile.nrx
2014 Dec 1 UploadPDS.nrx
2016 Aug 1 xml/
2016 Sep 2 db2conn.rexx
2016 Sep 2 db2query.rexx
2016 Sep 2 invdutil.rexx
2016 Sep 2 rvjcmf01.panel
2016 Sep 2 rvjcmf01.rexx
2016 Sep 2 rvjcmf01.skel
2017 Dec 1 ChangeFile1.nrx
2017 Dec 1 ChangeFile2.nrx
2017 Dec 1 ChangeFile3.nrx
2017 Jun 2 Kitchen.sink.bxml
2017 Oct .gitignore
2017 Oct AddFile.nrx
2017 Oct BaseChange.nrx
2017 Oct Calculator.nrx
2017 Oct CalculatorTest.nrx
2017 Oct ChangeReport.nrx
2017 Oct ChangedFiles.nrx
2017 Oct EbcdicTest.java
2017 Oct Expr.g4
2017 Oct ExprJoyRide.java
2017 Oct GetShal.nrx
2017 Oct HelloWorldMainTopic.java
2017 Oct HexPrint.nrx
2017 Oct InsertFile.nrx
2017 Oct JGitEBC.nrx
2017 Oct Migrate.nrx
2017 Oct MyPlayground.playground/
2017 Oct RSA.java
2017 Oct RetrieveFile.nrx
```

FIGURE 7: example 4



```
java org.vpad.extra.workpad.Workspace VM3
nrws> pipe command ls -laFTl | rexx specs 42-47 1 58-* 8 | specs 7-11 1 1-6 7 12-* 12 | sort | l
ocate /2019 Mar 2/ | console
2019 Mar 2 commandtest.njp
2019 Mar 2 testc2x.nrx
nrws> 0.157 s
```

FIGURE 8: example 5

Write your own Filters

So we have seen in the previous example that it is not too hard to make a simple pipeline out of things called 'device drivers' (such as *command*, for OS commands, '*'* for reading files on disk, and *literal*, for inserting literal strings into a pipeline, filters, and sinks). When a filter is not delivered in the standard set of stages, it is very easy to make one yourself in the NetREXX language. The model for this closely follows the way it is done with CMS Pipelines and Classic Rexx. Imagine, for the sake of argument (and a simple example⁵), that you have an assignment to quickly reverse a string.

```
1  /* BAGVENDT REXX -- Reverse the contents of lines in the pipeline */
2  signal on error
3  do forever
4    'peekto' data
5    'output' reverse(data)
6    'readto'
7  end
8  error: exit RC*(RC<>12)
```

And you would need to remember to call your filetype REXX instead of EXEC. The *peekto* reads the input but does not actually commit the read yet, so you can read it one more time with knowledge about the contents. The *output* pushes its argument back into the pipeline. The *readto* reads and commits the read so the line is really processed and we can go to the next one.

In NetREXX, that would be about the same, but for some small changes incurred by the object oriented model of NetREXX, which does not exist in Classic Rexx. Here *peekto()*, *readto()* and *output()* are method calls on the *stage* object. This will be made addressable by the import from *org.netrexx.njpipes.pipes*. (file: *bagvendt.nrx*)

```
1  import org.netrexx.njpipes.pipes.
2  class bagvendt extends stage
3  method run()
4    loop forever
5      line = Rexx peekto()
6      output(line.reverse())
7      readto()
8    catch StageError
9      rc = rc()
10   end
11  exit(rc*(rc<>12))
```

So that would look fairly familiar, and admittedly, a bit easier for us already well versed in NetREXX. We can test this by building a pipeline and running the filter on its own source:

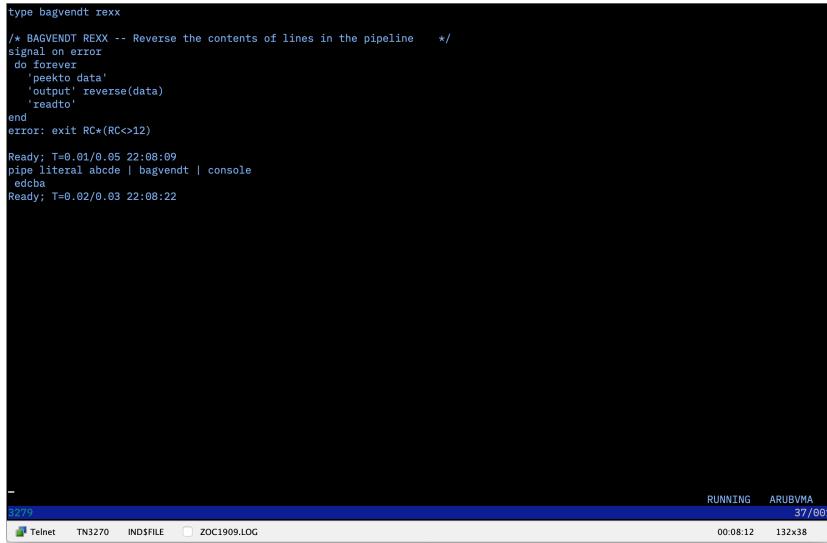
```
pipe "literal abcd | bagvendt | console"
```

⁵From the document CMS Pipelines Explained, by John P. Hartmann

If you have a CMS handy, that would be:

```
pipe literal abcd | bagvendt | console
```

on the first, Classic Rexx version of the filter - but the quoted version also works on CMS.

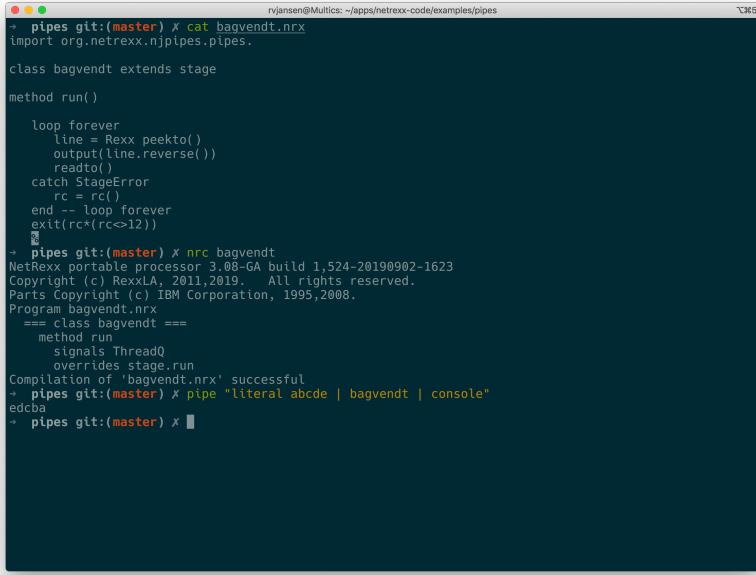


```
type bagvendt rexx
/* BAGVENDT REXX -- Reverse the contents of lines in the pipeline */
signal on error
do forever
  'peekto data'
  'output' reverse(data)
  'readto'
end
error: exit RC*(RC<>12)

Ready; T=0.01/0.05 22:08:09
pipe literal abcde | bagvendt | console
edcba
Ready; T=0.02/0.03 22:08:22
```

The screenshot shows a terminal window with a black background and white text. It displays the source code of the bagvendt filter, its execution, and the reversed output. The window title is 'Telnet TN3270 INDSFILE ZOC1909.LOG'. At the bottom, there's a status bar with 'RUNNING ARUBVMA 37/001', a timestamp '00:08:12', and a resolution '132x38'.

FIGURE 9: BAGVENDT under VM/CMS



```
rvjansen@Multics: ~/apps/netrex-code/examples/pipes
→ pipes git:(master) X cat bagvendt.nrx
import org.netrex.njpipes.pipes.

class bagvendt extends stage

method run()
  loop forever
    line = Rexx peekto()
    output line.reverse()
    readto()
  catch StageError
    rc = rc()
  end -- loop forever
  exit(rc*(rc<>12))
→ pipes git:(master) X nrc bagvendt
NetRexx portable processor 3.08-GA build 1,524-20190902-1623
Copyright (c) RexxLA, 2011,2019. All rights reserved.
Parts Copyright (c) IBM Corporation, 1995,2008.
Program bagvendt.nrx
  === class bagvendt ===
  method run
    signals ThreadQ
    overrides stage.run
  Compilation of 'bagvendt.nrx' successful
→ pipes git:(master) X pipe "literal abcde | bagvendt | console"
edcba
→ pipes git:(master) X
```

The screenshot shows a terminal window with a dark background and light text. It displays the source code of the bagvendt.nrx filter, its compilation, and its execution. The window title is 'pipes git:(master) X'. At the bottom, there's a status bar with 'RUNNING ARUBVMA 37/001', a timestamp '00:08:12', and a resolution '132x38'.

FIGURE 10: bagvendt.nrx under NetREXX

More advanced Pipelines

Admittedly, the examples in the previous chapters could have been done with Unix pipes or at least with incorporation of stream utilities like awk or sed.

To get a good idea of what can be done with Pipelines for NetREXX, look at the tasktest pipe in the examples directory. It⁶ implements the shell of a multitasking server - using about eight stages. The file examples/tcptask.njp contains an example of this technique being used.

```
1 --tasktest.njp
2
3 pipe (tasktest stall 2000 -gen)
4
5 literal 0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T |
6 dup 2 |
7 split |          -- supply work for task stage
8
9 ptimer |
10 a: deal secondary ? -- send work to task stage requesting work
11 b: faninany |
12 elastic |        -- buffer requests to so no deadlocks
13 ptimer |
14
15 a: |
16 copy |           -- buffer work so no deadlocks
17 task 1 |         -- worker task 1
18 b: ?
19
20 a: |
21 copy |           -- worker task 2...
22 task 2 |         -- worker task 2...
23 b: ?
24
25 a: |
26 copy |           -- worker task 3...
27 task 3 |         -- worker task 3...
28 b:
```

Before discussing this example in-depth, we need to go into some more basic concepts.

⁶using code from Melinda Varians 'Cramming for the Journeyman Plumber Exam' paper

Device Drivers

Pipelines for NetREXX contains the following device drivers:

<	read from a file
>	write to a file (which is overwritten if it exists)
»	append to a file (which is created if it does not exist)
diskr	read from a file
diskw	write to a file (which is overwritten if it exists)
diska	append to a file (which is created if it does not exist)
diskslow	read, create or append to a file
array	manipulate arrays
arraya	manipulate arrays
arrayr	manipulate arrays
stem	manipulate stems
stema	manipulate stems
stemr	manipulate stems
vector	manipulate vectors
vectora	manipulate vectors
vectorr	manipulate vectors
var	read or set a variable in a NetREXX program
zip	compress a set of files (0 or more) into a zip archive
unzip	decompress a set of files (0 or more) from a zip archive
listzip	list a zip file directory
console	read from, or write to a terminal (window)
hole	destroy data
delay	suspend stream
literal	write the argument string
strliteral	write the argument string
sqlselect	select from any jdbc source
xrange	write a character range

Record Selection

Various stages can select records and work on data in the pipeline. These are stages called select, sort, specs, locate, etcetera. For a complete description we refer to the IBM Pipelines documentation.

These are the main selection stages supported in Pipelines for NetREXX:

between	selects records between labels
drop	discard records from the beginning or the end of a file
find	select lines
strfind	select lines
frlabel	select records from the first one with leading string
strfrlabel	select records from the first one with leading string
inside	select records between labels
locate	select records between labels
nfind	select lines using xedit nfind logic
strnfind	select lines using xedit nfind logic
nlocate	select lines without a string
notinside	select records not between labels
outside	select records not between labels
pick	select records that satisfy a relation
take	select records from the beginning or the end of a file
tolabel	select records to the first one with leading string
strtolabel	select records to the first one with leading string
unique	discard or retain duplicate lines

Filters

buffer	buffer records
chop	truncate the record
join	join records
pad	expand short records
split	split records relative to a target
change	substitute contents of records
specs	rearrange contents of records
xlate	transliterate contents of records
copy	copy records
count	count lines, words and bytes
dup	duplicate the object
reverse	reverse contents of records
timestamp	prefix date and time to records
append	put output from device driver after data on the primary input
casei	run selection stage in a case-insensitive manner
not	run stages with output streams inverted
prefix	Blocks its primary input and executes stage supplied as an argument
zone	run selection stage on subset of input record
elastic	buffer sufficient records to prevent stall
fanin	concatenate streams
faninany	copy records from whichever input stream has one
gate	pass records until stopped
juxtapose	preface record with marker
overlay	overlay data from input streams
command	issue a command and write response to pipeline

Other Stages

query	check version and level of Pipelines for NetREXX
--------------	--

--	insert comments into a pipeline
----	---------------------------------

comment	insert comments into a pipeline
----------------	---------------------------------

Multi-Stream Pipelines

One of the defining differences with Unix pipes is the possibility to define multi-stream pipelines. The selection stages from the previous chapter all have *secondary streams*. What the selection parameters have discarded, *seem to have discarded*, is in reality not gone. In fact, Pipelines for NetREXX throws very little away during execution.

The way to use the not-selected part of the data through these secondary streams is explained in this chapter; it is this capacity that constitutes the freedom to work with many different streams in one pipeline; where Unix pipes are limited to not very much more than stdin, stdout, stderr – Pipelines for NetREXX enables the user to define as many streams as necessary to accomplish the task at hand in an efficient manner.

Let us look at a simple selection like the following:

```
1 pipe literal foo bar baz frob frobnitz frobotzim | split | locate /oo/ |
2 console  
  
foo
```

The string that makes it through the selection that is done by the *locate* is 'foo' - it is the only one that is captured by the /oo/ filter.

The rest of the words is not gone, however, and we can use these in further processing by using the secondary stream that *locate* provides.

To prepare for this, we give the secondary stream a name by providing a label for it, we call it, in absence of any creativity, *rest*⁷. Also, we send the selected output, foo into a *hole* stage, where it disappears.

```
1 pipe literal foo bar baz frob frobnitz frobotzim | split | rest: locate /oo/ |
2 hole
```

As predicted, there is no output. To get to the rest of the words, unselected by *locate*, we connect the secondary output stream to a new pipe, using the '?' (the default pipe-end character) like this:

```
1 pipe literal foo bar baz frob frobnitz frobotzim | split | rest: locate /oo/ |
2 hole ? rest: | console
```

The output is now:

```
bar
baz
frob
frobnitz
frobotzim
```

⁷often, you will see it being called 'a'

Instead of sending the original output into a black *hole*, we could have also gone further with it, and, for example, reverse it:

```
1 pipe literal foo bar baz frob frobnitz frobotzim | split | rest: locate /oo/ |
2 reverse | console ? rest: | console
```

The output is now:

```
oof
bar
baz
frob
frobnitz
frobotzim
```

Likewise, we can specify more filter stages in the second, attached pipeline, and bifurcate the pipeline even further.

```
1 pipe literal foo bar baz frob frobnitz frobotzim | split | rest: locate /oo/ |
2 reverse | console ? rest: | locate /botzim/ | console
```

The output is now:

```
oof
frobotzim
```

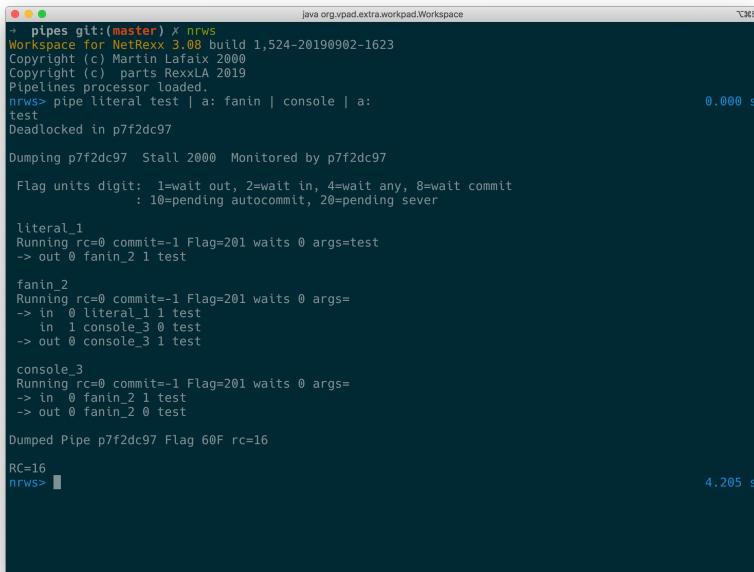
It is good to define and implement secondary streams when you write your own stages.

Pipeline Stalls

With multistream pipelines a new problem sometimes rears its head - a *Pipeline stall*, also called *deadlock*. This happens when stages wait for input that cannot be delivered, in a way that ensures that it cannot be delivered.

Pipes for NetREXX detects deadlocks and outputs information to allow you to fix the problem. Consider the following session:

```
1 pipe literal test | a: fanin | console | a:
```



The screenshot shows a terminal window titled "java org.vpad.extra.workpad.Workspace" running on a Mac OS X system. The window displays the following NetREXX session output:

```

-> pipes git:(master) x nrws
Workspace for NetREXX 3.08 build 1,524-20190902-1623
Copyright (c) Martin Lafalx 2000
Copyright (c) parts RexxLA 2019
Pipelines processor loaded.
nrws> pipe literal test | a: fanin | console | a:
test                                         0.000 s
Deadlocked in p7f2dc97

Dumping p7f2dc97 Stall 2000 Monitored by p7f2dc97
Flag units digit: 1=wait out, 2=wait in, 4=wait any, 8=wait commit
: 10=pending autocommit, 20=pending sever

literal_1
Running rc=0 commit=-1 Flag=201 waits 0 args=test
-> out 0 fanin_2 1 test

fanin_2
Running rc=0 commit=-1 Flag=201 waits 0 args=
-> in 0 literal_1 1 test
  in 1 console_3 0 test
-> out 0 console_3 1 test

console_3
Running rc=0 commit=-1 Flag=201 waits 0 args=
-> in 0 fanin_2 1 test
-> out 0 fanin_2 0 test

Dumped Pipe p7f2dc97 Flag 60F rc=16
RC=16
nrws>                                         4.205 s

```

FIGURE 11: Deadlock detection

We can see that there are three stages in the Running state. None have any return codes set. The Flags tell us that all the stages are waiting for an output to complete.

The '-' show which stream is selected. From this we can see console_3 is trying to output to fanin_2. Unfortunately fanin_2 is waiting for output on stream 0 to complete, it cannot read the data waiting on in stream 1. Hence the stall.

The strings after *Dumping* and *Monitored by* are the autogenerated class names. When you name your pipelines with precompiled pipes yourself, the names you have given them will be displayed here.

When a stream has data being output, there is a boolean flag following the name of the stage the stream is connected to. This tracks the peek state of the object. For an output

stream, true means the following stage has peeked at the value. With input streams, the current stage has seen the value when its true.

When a stage is multithreaded, like elastic, you can get flags of 3 or 5. This means that threads are waiting on output and read, or output and any. When using multithreaded stages, only one thread should use output unless it is serialized using protected or synchronized blocks.

When a stage has a pending sever or autocommit, flag bits are set too.

Differences with CMS Pipelines

The goal of this implementation is to be as close as possible to the the CMS version of Pipelines. A few differences are unavoidable.

- The character set is Unicode and not EBCDIC, as Unicode is the character set of the underlying Java platform
- As shells are different, many 3270 related stages are not implemented
- Pipes need to be quoted on the Windows and Unix command lines; the Workspace for NetREXX (*nrws*) environment is an exception to this rule
- The mainframe is record-oriented in many stages, Pipelines for NetREXX does an approximation of this
- Pipelines on the mainframe is an interpreted language with components as the scanner and the dispatcher; the NetREXX version is compiled to Java .class files by *pipc*, the pipes compiler, and dispatched as threads by the JVM.
- The mainframe pipes dispatcher is not multiprocessor enabled. In Pipelines for NetREXX all tasks (stages) are dispatched over all available processors in parallel.
- The fact that pipes run from NetREXX implies that they can be used in Java source. In previous releases there was more direct support for this; this has lapsed due to changes in the way a java toolchain works. This support can be restored in future releases.
- To put the content of a NetREXX variable in a pipe specification in a NetREXX program, there is a {} mechanism. In CMS the pipe would be quoted in the Rexx source and you would unquote sections to get a similiar effect.

How to use a pipe in a NetREXX program

This shows how to use a pipe in a NetREXX program:

```
1 class testpipe
2
3     method testpipe(avar=Rexx)
4
5         F = Rexx 'abase'
6         T = Rexx 1
7
8         F[0]=5
9         F[1]=222
10        F[2]=3333
11        F[3]=1111
12        F[4]=55
13        F[5]=444
14
15        pipe (apipe stall 1000 )
16            stem F | sort | prefix literal {avar} | console | stem T
17
18        loop i=1 to T[0]
19            say 'T['i']=T[i]
20        end
21
22    method main(a=String[]) static
23
24        testpipe(Rexx(a))
```

A couple of things can be seen in this example. First that it is simple to pass NetREXX variables to pipes using *stem*. Also look at the phrase *{avar}*. It passes the NetREXX variable's value to the stage at runtime. In CMS the pipe would be quoted and you would unquote sections to get a similiar effect.

Another thing to note is that the pipe extraction program is fairly smart. It detects when pipes takes several lines. As long as there are stages, or the current line ends with a stage-sep or stageend character, or the next line starts with a stagesep or stageend character. It gets added to the pipe.

The *arg()*, *arg(rexx)* or *arg(null)* methods get the arguments passed to a stage or pipe. To get the complete rexx string of an argument use *arg()*. To get the nth word of a rexx argument use *arg(n)*. When using pipes in netrexx code you can use *arg('name')* to get the named argument. If the class of the argument is not rexx use *arg(null)* to get the object.

In .njp files you can use *avar* phrase actually just shorthand for *arg('avar')*. The following example shows what has to be done in a stage to access the rexx variables passed by VAR, STEM and OVER. The real over stage is a bit more complete.

```
1 -- over.nrx
2 class over extends stage final
3
4     method run() public
5         a = getRexx(arg())
6         loop i over a
```

```

7     output(a[i])
8   catch StageError
9     rc = rc()
10    end
11
12  exit(rc*(rc>>12))

```

The getRexx method is passed the name of a string by the pipe. In the previous example it would be passed A and would return an Object pointer to A in testpipe. If you wish to replace a stream this can be done using connectors. For example look at the following fragment:

```

-- examples\calltest.njp
pipe (callt1) literal test | calltest {} | console

1 import org.netrexx.njpipes.pipes.
2
3 class calltest extends stage final
4
5 method run() public
6
7   do
8     a = arg()
9
10    callpipe (cp1) gen {a} | *out0:
11
12    loop forever
13      line = peekto()
14      output(line)
15      readto()
16    end
17
18  catch StageError
19    rc = rc()
20  end
21
22  exit(rc*(rc>>12))

```

Running the callt1 pipe with an argument of 10 would pass the 10 to calltest via `arg()`. Then cp1's gen stage would be passed 'a' which is set to 10. Since gen generate numbers in sequence, the console stage of callt1 would get the numbers from 1 to 10. Now cp1 ends and calltest's output stream is restored and calltest unblocks and reads the the literal's data 'test' and passes it to console.

The use of `*` only works when compiling from .njp files. It will not work from the command line. The njpipes compiler recognizes connectors as labels with the following forms:

```

*in:
*inN:
*out:
*outN

```

When N is a whole number, the connector connects input or output stream N of the stage with the connector. When the label `*in` or `*out`, the connector connects the stages's current input or output stream with the connector. This is used instead of `:` due to the way the compiler/preprocessor works. If you do not want the stage to wait for the called pipe to complete you can use addpipe. Here is an example.

```

1 -- similar to examples\addtest.njp
2
3 a = 100

```

```

4     b = 'some text for literal'
5
6     addpipe (linktest) literal {b} | dup {a} | *in0:
7
8     loop forever
9         line = Rexx readto()
10        catch StageError
11    end

```

readto() will get 'some text for literal' one hundred times.

A quick aside. When writing stages remember that njPipes moves objects through pipes. Use 'value = peekto()' instead of 'value = rexx peekto()' when ever possible. Some of the supplied stages pass objects with classes other than rexx and forcing rexx will cause class-CastExceptions. If a stage needs a rexx object try using the rexx stage modifier to attempt to convert the object. Feel free to expand this stage, but please send me the updated version.

Serious stage writers will probably want to take a good look at the methods defined in the NetREXX source package `org.netrexx.process.njpipes.stages`. There you will find various methods for parsing ranges. You will also find the stub for the stageExit compiler exit. It can be used to produce 'on the fly' code at compile time. You can also use it to change the topology of the unprocessed part of the pipe. The major use is to allow implementations of stages like prefix, append or zone. Its also used to produce better performing stages, for an example see specs. The compiler also queries the `rexxArg()` and `stageArg()` methods. If your stage expects objects of class Rexx as arguments `rexxArg()` should return the number of variables expected. If your stage expects a stage for an argument, `stageArg()` should return the word position of the stage.

TCP/IP Networking using Pipes for NetREXX

As the built-in stages all work on data that is dispatched through the pipeline, irrespective of which device driver is used, it is also convenient to do network programming using a set of pipelines.

The *tcplisten* stage can be used as a network device driver, as in CMS, but limited to specification of the port and a timeout value. Below an example of how to implement a sample TCP/IP client/server application.

```
1 -- one shot tcip server
2
3 pipe (tcperv stall 60000 debug 0 )
4   tcplisten 1958 timeout 15000 | tcpexample
5
6 -- one shot tcip requestor
7
8 pipe (tcpreq stall 60000 debug 0 )
9   random {} |
10  specs *-* 1 ,\n, next |
11  tcpclient deblock c localhost 1958 timeout 10000 linger 500 oneresponse |
12  rexz to console
13
14 -- a single tasking server
15
16 options binary
17 import org.netrexx.njpipes.pipes.
18 class tcpexample extends stage
19
20 method run() public
21
22 loop forever
23
24 peekto()
25
26 callpipe (tcplog stall 15000 debug 0)
27   *in0: |
28   take first 1 |
29   console |
30   f: fanin |
31   tcpdata timeout 10000 deblock C oneresponse |
32   elastic |
33   insert /\n/ after |
34   f:
35
36 catch StageError
37   rc = rc()
38 end
39
40 exit(rc*(rc<>12))
```

This example needs to be compiled with the pipes compiler, see *TCP/IP Client/Server compile*, which yields the classes *tcperv* and *tcpreq*, for the server and the requester component.

Now we can start the generated pipelines each in their own shell window. As can be seen in *TCP/IP server*, the class keeps waiting on connections on port 1958 - which is

```

rvjansen@Multics: ~/apps/netrexx-code/examples/pipes
→ pipes git:(master) x pipc tcpexample.njp
pipe (tcpserv stall 60000 debug 0 ) tcplisten 1958 timeout 15000 | tcpexample
pipe (tcpreq stall 60000 debug 0 ) random arg() | specs *-* 1 ,\n, next | tcpclient deblock c loc
alhost 1958 timeout 10000 linger 500 oneresponse | rexxy to console
callpipe (tcpexample_tcplog stall 15000 debug 0) *i_A0: | take first 1 | console | f: fanin | tcp
data timeout 10000 deblock C oneresponse | elastic | insert /\n/ after | f:
→ pipes git:(master) x

```

FIGURE 12: TCP/IP Client/Server compile

arbitrary, but specified in the pipeline source.

```

rvjansen@Multics: ~/apps/netrexx-code/examples/pipes
→ pipes git:(master) x java tcpserv
Socket[addr=/127.0.0.1, port=63127, localport=1958]
Socket[addr=/127.0.0.1, port=63164, localport=1958]
Socket[addr=/127.0.0.1, port=63195, localport=1958]

```

FIGURE 13: TCP/IP server

In another window, we can start the *TCP/IP requestor*, which when given port 1958 as argument, connects to the server, and displays a series of random numbers that is sent to it.

```

rvjansen@Multics: ~/apps/netrexx-code/examples/pipes
→ pipes git:(master) x java tcpreq 1958
1001001884
235454313
417317619
377575257
971252730
261029274
1502933356
735098630
1885452794

```

FIGURE 14: TCP/IP requestor

Note that the stage *tcpexample* from the *tcpserver* pipeline is a custom stage that is written in this *tcpexample.njp* file.

Selecting from databases with Pipelines for NetREXX

Using the built-in *sqlselect* stage you can select data, using SQL, from any jdbc source available.

An *sqlselect.properties* file is needed to define the jdbc parameters like the driver to use, the url of the data source and other arguments, like a password and tracing options, if needed.

The file looks like this:

```
jdbcdriver=org.sqlite.JDBC  
url=jdbc:sqlite:flightroute-iata.sqb
```

This is all that is needed for an sqlite database containing flight data. A simple select * can then be done with the following pipeline:

```
| pipe literal * from FlightRoute where flight = 'KLM765' | sqlselect | console
```

This yields the following output:

```
FLIGHT--ROUTE--UPDATETIME--  
KLM765 AUA-BON-AMS 1494132448
```

Note that from the command line, the quotes around the pipe specification and the literal string in the SQL statement should be opposite, while when the pipeline is issued from the Workspace for NetREXX, the pipeline does not have to be quoted, but the sql string needs double quotes instead of the - for SQL statements- normal single quotes.

The Pipes Runner

The pipes compiler is used in both precompiled and directly executed pipelines. When you directly execute a pipeline from the commandline or from the *nrws* NetREXX workspace, the process is optimized to not persist generated NetREXX, Java and Class files to disk before execution, the whole process runs from memory. The Pipes Runner uses the Pipes Compiler for this purpose, and as such misses the options for persistence⁸.

The *pipe* command alias start the Pipes Runner, which is a command processor that can execute a pipe from the command line in an OS shell, the OS being Windows, Linux or macOS⁹.

A pipe can be run with options prepended within parentheses, like this:

```
| pipe '(test1 sep ! stall 2000 debug 63) literal abcde ! console'
```

The following options are available:

pipename	Specify the name of the generated class file. This can be useful for debugging purposes but is not mandatory when running a pipe. An unnamed pipe receives a generated unique name. This option needs to go first.
sep	The default stage separator is the (pipe) character; this can be overridden with the sep option; a pipe called test1 which uses an exclamation mark as separator character, needs the options (test1 sep!).
debug	The debug option specifies a bitmask for debugging the execution of a pipe; (debug 63), for example, generates a rather complete debugging trail.
end	The default pipe end character is the ' ?' (question mark), which can be overridden here. Note that the backslash, which is an obvious pipe end character for the z/VM 3270 interface, is not a good choice for Windows and Unix shells.
stall	The duration in number of seconds of a pipe stall (or deadlock) detection cycle.

⁸But specifying them will not generate an error

⁹this is a non-exhaustive list of operating systems

The Pipes Compiler

The purpose of precompiling a pipeline specification is to produce a .class file for the JVM that can be run independently and on different machines; only the JVM and the NetRexxC.jar or the NetRexxF.jar are required to run a precompiled pipe. A set of pre-compiled pipes can be shipped as an application.

When precompiling pipes, there are options to save and view the generated NetREXX, Java and JVM Class files. A precompiled pipe has the advantage that it can be executed over and over in an application, without the need to compile it every time; the performance savings are accumulative in this scenario.

The following options can be used on the *pipc* command, in addition to the ones specified in the previous chapter for the Pipes Runner:

-gen Generate the NetREXX source file. The pipeline needs a name.

-keep Keep the Java source which is generated from the NetREXX source.

Example:

```
1 pipe (testpipe -gen -keep)
```

This will generate the NetREXX source as well as keep the java source.

Built-in Stages

This section describes the set of built-in stages, i.e. the ones that are delivered with the downloadable open source package. These stages are directly executable from the NetRexxC.jar file or the NetRexxF.jar file (the latter contains a Java compiler for use on JRE-only systems); also, the source of these stages is delivered in the NetREXX source repository. This repository can be checked out at

```
git clone https://git.code.sf.net/p/netrexx/code netrexx-code
```

The source of the stages is in directory

```
netrexx-code/src/org/netrexx/njpipes/stages
```

--	Pipelines for NetRexx only; delegates to comment
<	Implemented as in CMS; delegates to diskr .
>	Implemented as in CMS; delegates to diskw .
>>	Implemented as in CMS; delegates to diska .
?	Not implemented. Help.
3270bfra	Not implemented. Old terminal support.
3270enc	Not implemented. Old terminal support.
abbrev	>>--ABBREV--+-----+----->< +--word--+-----+-----+ +--number--+-----+-----+ +--ANYCASE--+ Pipes for NetRexx only
aftfst	Not implemented. Open file information
ahelp	Not implemented. Author's help
all	Not implemented. Select all records containing a specified string or strings determined by an expression
apldecode	Not implemented. Old APL language
aplencode	Not implemented. Old APL language
append	>>--APPEND-- <i>string</i> --><
array	Pipes for NetRexx
arraya	Pipes for NetRexx
arrayr	Pipes for NetRexx
arrayw	Pipes for NetRexx
asatomc	Not implemented. Old printer control
asmcont	Not implemented. Assembler language
asmfind	Not implemented. Assembler language
asmnfind	Not implemented. Assembler language
asmxpnd	Not implemented. Assembler language
between	>>--BETWEEN--+-----+--/ <i>string</i> /---+--/ <i>string</i> /---+----->< +--ANYcase--+ +--n-----+ +--n-----+
bfs	Not implemented. Read Bit Stream File
bfsdirectory	Not implemented. Read Bit Stream Directory
bfsquery	Not implemented. Read Bit Stream current Directory
bfsreplace	Not implemented. Bit Stream file
bfstate	Not implemented. Read Bit Stream file
bfsxecute	Not implemented. Read Bit Stream file
block	Not implemented. See fblock. Reformat records
buffer	>>--BUFFER--+-----+----->< +--n--+-----+--+ +--/ <i>string</i> /--+
buildscr	Not implemented. Old terminal
casei	>>--CASEI-- <i>stage</i> --><
change	+--1-*-----+ >>--CHANGE--+-----+-----+-----> +--ANYcase--+ +-- <i>inputRange</i> -----+ <-----+ +-(---+range+---)-+ >--+-/ <i>string1</i> / <i>string2</i> /---+-----+----->< +--/ <i>string</i> /--/ <i>string</i> /--+ +--numorstar--+

chop	+--80-----+ ->---+-CHOP----+-----+ ---TRUNCate-+ ---column---
cms	Not implemented. OS dependent.
collate	Not implemented. Match records from 2 streams.
combine	Not implemented. Joins bit-oriented streams.
command	>>---COMMAND---+-----+ ---string---+ -- input stream 0 is for commands -- input stream 1 is stdin -- output stream 0 is stdout -- output stream 1 is the return code -- output stream 2 is stderr
configure	Not implemented. Create specifications for CMS Pipelines in z/VM.
compare	>>---COMPARE---+-----+ +-TRINARY-+ (1) +-PAD SPACE-+ +-BINARY---+ (2) +-PAD-Xorc---+ <-----+ +-ANY DString-----+---+ (4) (5) +-EQUAL DString----+ (4) +-LESS DString-----+ (3) (4) +-MORE DString-----+ (3) (4) +-NOTEQUAL DString-+ (4)
	(1) -1 = Primary is shorter/less, 0 = equal, 1 = Secondary is shorter/less (2) 0 = equal, 1 = not equal (3) Primary is LESS/shorter (or MORE/longer) than secondary (4) DStrings can use any of the following escapes (or the lowercase) for the unequal situation: \<count> for the record number, \<byte> for column number \<primary> for the primary stream record \<secondary> for the secondary stream record \<least> for then stream number that is shortest, -1 if equal \<most> for the stream number that is longest, -1 if equal 5. Equal or not, this DString precedes any of the others. Pipes for NetRexx only.
configure	Not implemanted. OS dependent
console	>>---+-CONSOLE---+-----+ +-TERMINal-+ ---EOF---/string/---+ +-NOEOF-----+
copy	>>---COPY-----+ copy from input stream to output without delaying the stream. See elastic for a more generic way to do this.
count	>>---COUNT---v---+-----+ ---CHARACTERs---+-----+ +-WORDS-----+ +-LINES-----+ +-MINline-----+ +-MAXline-----+
cp	Not implemented. OS specific.

crc	Not implemented. Compute a checksum
c14to38	Not implemented. Old printer.
dam	Do no pass any objects thru secondary streams until an object appears on the primary input stream. The primary output stream must not be connected. Pipes for NetRexx only.
dateconvert	Not implemented. Timestamp conversion and validation.
deal	<pre>+--STOP--ALLEOF-----+ >>--DEAL--+ +--STOP--+ALLEOF---+ +--ANYEOF--+ +--n-----+ +--SECONDARY--+ +--RELEASE--+ +--KEY--inputRange--+ +--STRIP--+ +--STREAMid--inputRange--+ +--STRIP--+ +--STRIP--+</pre> <p>Since Java dispatches the stage threads. Deal may not see a sever immediately, as the severing thread can get multitasked. This can make options like 'ANYEOF' work in unexpected ways.</p>
deblock	<pre>+--FIXED--n--+ +--PAD--xorc--+ >>--DEBLOCK--+ +--C-----+ +--J-----+ +--TERMINATE--+ +--EOF----+ +--CRLF-----+ +--LINEND xorc-----+ +--STRING--/string/--+</pre> <p>?? System.getProperty('line.separator') ??</p>
delay	Not implemented. Wait for clock time
dict	Pipes for NetRexx only.
dicta	
dicta	
dictw	
disk	As in CMS, equivalent to diskr (Pipes for NetRexx Only) or <.
diska	Appends records on its input stream to the end of the supplied file, the file is created if its does not exist.
diskr	
diskslow	
diskw	
drop	<pre>+--FIRST--+ +--1--+ >>--DROP--+ +--LAST--+ +--n--+ +--BYTES--+ +--*--+</pre>
dup	<pre>+--1----+ >>--DUP--+ +--n----+ +--*----+ +-- -1----+</pre>
elastic	>>--ELASTIC--><

ems	Not implemented. OS specific.
escape	Not implemented. Insert escape characters so special characters are treated a
fanin	>>--FANIN--+----->x <-----+ +---+--stream---+
faninany	>>--FANINANY----->x
fanout	+--STOP--ALLEOF-----+ >>--FANOUT--+----->x +--STOP---+--ANYEOF---+ +--n-----+
fblock	>>--FBLOCK--number-----+ +--xorc---+
file	As in CMS. Synonym of disk
filea	Pipes for NetRexx only. Synonym of diska
fileback	Not implemented. Read a CMS file backwards.
filefast	Not implemented. Read or write a CMS file.
filer	Pipes for NetRexx only. Synonym of diskr
filerand	Not implemented. Read specific records from a CMS file.
fileslow	Synonym for diskslow .
fileupdate	Not implemented. Change records in a CMS file.
filew	Pipes for NetRexx only. Synonym of diskw
find	>>---+--FIND--+----->x +--string--+
fmtfst	Not implemented. OS specific
frlabel	>>---+--FRLABEL--+----->x +--string--+
frtarget	>>---+--FRTARGET----+--stage--+----->x +--FROMTARGET--+ +--operands--+
fullscreen	Not implemented. Old terminal.
fullscrq	Not implemented. Old terminal.
fullscrs	Not implemented. Old terminal.
gate	>>--GATE--+----->x +--STRICT--+
gather	Not implemented. Read records in specified order.
getfiles	>>--GETfiles----->x
getovers	Input stream 0 should contain rexx objects. The getovers stage will output will output the index and contents of the stem on stream 0. If output stream 1 is connected, the root is placed there. Any severed streams will cause then stage to exit. Passing a non rexx object will cause the stage to exit with return code 13 Pipes for NetRexx only
getstems	Input stream 0 should contain rexx objects containing stems. The getstems stage will output the contents of the stem on stream 0. If output stream 1 is connected, the root is placed there. Any severed streams will cause then stage to exit. Passing a non rexx stem object will cause the stage to exit with return code 13 Pipes for NetRexx only.
hash	Pipes for NetRexx only. Synonym of dict
hasha	Pipes for NetRexx only. Synonym of dicta
hashr	Pipes for NetRexx only. Synonym of dictr
hashw	Pipes for NetRexx only.

help	Not implemented. Help for CMS Pipelines.
hole	>>--HOLE--><
hostbyaddr	Not implemented. Resolve IP address to domain and host name.
hostbyname	Not implemented. Resolve domain name to IP address.
hostid	Not implemented. Writes default IP address.
hostname	Not implemented. Writes host name of the TCP/IP system.
iebcopy	Not implemented. OS dependent.
immcmd	Not implemented. OS dependent.
insert	<pre>+--BEFORE--+ >>--INSERT--/string/---+-----+----->< +--AFTER---+ insert a string into a record. Will be much more efficient than specs especially if the input is a Byte[] Pipes for NetRexx only.</pre>
inside	>>--INSIDE--/string1/---+/string2/---+----->< +--n-----+
instore	Not implemented. Read records into storage.
ip2socka	Not implemented. Convert IP address to special hex.
ispf	Not implemented. OS dependent.
join	<pre>+--1-----+ >>--JOIN--+-----+-----+-----+-----+----->< +--n-----+ +--/string/-+ +-maxlength(1)--+ +--*-----+ +--KEYLENgth--n--+</pre> <p>(1) The <i>maxlength</i> operand cannot be entered without specifying other operands.</p>
joincont	<pre>+--TRAILING-----+ >>--JOINCONT--+-----+-----+-----+-----+-----+----->< +--ANYCase--+ +--NOT--+ +--RANGE--range--+ +--ANYof--+ +--LEADING-----+</pre>
juxtapose	>>--JUXTAPOSe--><
ldrtbls	Not implemented. Assembler: precompiled.
listpds	Not implemented. OS dependent.
literal	<pre>+--{-object-name-}--+ >>--+-LITERAL---+-----+-----+----->< +--string---+ +--PREFACE---+</pre>
locate	>>--LOCATE--+-----+-----+-----+-----+-----+--> +--ANYcase--+ +--inputRanges--+ +--ANYof--+ +--/string/-+

lookup	in stream 0 are detail records in stream 1 are master records in stream 2 adds to masters in stream 3 delete from masters
	out stream 0 are matched records out stream 1 are unmatched detail records out stream 2 are unmatched or counted master records out stream 3 deleted masters out stream 4 duplicate masters out stream 5 unmatched master deletes
	<p>lookup does not consider an unconnected output stream an error. It does propagate EOFs from output streams.</p>

To increase performance reorder the 'when type=' in method out so the type you use is first in the list and recompile the stage.

```
>>--LOOKUP--+-----+-----+-----+-----+-----+----->
    +-COUNT--+  +-ANYCASE--+  +-AUTOADD--+  +-BEFORE--+
>-+-----+-----+-----+-----+-----+-----+----->
    +-KEYONLY--+  +-SETCOUNT--+  +-INCREMENT--+  +-TRACKCOUNT--+
>-+-----+----->
    +-inputRange--+-----+
    +-inputRange--+
>-+-----+----->x
    +-DETAIL MASTER-----+
    +-DETAIL ALLMASTER PAIRWISE--+
    +-DETAIL ALLMASTER-----+
    +-DETAIL-----+
    +-MASTER DETAIL-----+
    +-MASTER-----+
    +-ALLMASTER DETAIL PAIRWISE--+
    +-ALLMASTER DETAIL-----+
    +-ALLMASTER-----+
```

maclib	Not implemented. OS dependent.
mctoasa	Not implemented. Old printer.
mdiskblk	Not implemented. OS dependent.
members	Not implemented. OS dependent.
merge	Not implemented. Merge up to 10 input streams.
nfind	>>--NFIND--+-----+----->x +-string--+
ninside	Not implemented. Alias for notinside .
nlocate	>>--NLOCATE--+-----+-----+-----+-----+-----+>x +-ANYcase--+ +-inputRanges--+ +-ANYof--+ +-/string/-+
noEofBack	Pipes for NetRexx only. Do not propagate eof back thru this stage.
nop	Pipes for NetRexx only.
not	>>--NOT--stage--+-----+----->x +-operands--+

runpipe	Not implemented.
scm	Not implemented. Clean up comments in REXX and C.
serialize	{class} if class is specified deserialize input to objects of this type other Pipes for NetRexx only. For some reason readObject does not like more than one object network in its examples/sertest.njp
snake	Not implemented. Reformat one or more records into matrices.
socka2ip	Not implemented. Convert special 16-byte hex to IP address.
sort	sort objects: sort {{<Class>}} {<size>} {A D} {IRange} Where <Class> is a sortClass. The default is: Rexx and <size> is the maximum default. Pipes for NetRexx quirks.
sortClass	This is an interface class to allow a generic sort routine to handle objects implemented by sortRexx, which will sort rexx objects. Pipes for NetRexx only.
sortRexx	An implementation for sortClass for rexx objects. Part of the logic to generate sortClass stages is in sort's stageExit. Pipes for NetRexx only.
specs	Massage the data. Selections via start.length start-end, start-* and -from_e These selections can also be prefixed with 'word'. The conversion functions upper, lower, b2x, d2x, x2b and x2d are implemented format options left, right and center. The stop, select, read, readstop, pa work too. The number, TODclock and many other conversion functions are not implemented. <pre>>>>-SPECs--+-+-----> +--STOP--+-ANYEOF---+- +-n-----+ + V >>>--+ Group -+----->< +--READ-----+ +--READSTOP-----+ +--WRITE-----+ +--SELECT--+-streamnum--+ +--streamid---+ +--PAD---+char-----+ +--hexchar--+ +--BLANK----+ +--SPACE----+ +---WORDSEparator---+char-----+ +--WS-----+ +--hexchar--+ +--FIELDSEparator--+ +--BLANK----+ +--FS-----+ +--SPACE----+ </pre> <p>Group:</p> <pre> -- Input -- Conversion -- Output -- Alignment ----- </pre> <p>Input: <pre> ---+Words-----wnumberrange-----+----- +-Fields-----fnumberrange-----+ +-cnumberrange-----+ +-/string/-----+ +-Xhexstring-----+ +-Hhexstring-----+ +-Bbinstring-----+ - +-FROM--1-----+ +-BY--1-----+ +-RECN0-----+-----+-----+-----+ - +-FROM--fromnum--+ +-BY--bynum--+ +-TODclock-----+</pre> </p>

Specs	Conversion:
(cont)	<pre> ---+-----+-----+ +-STRIPI--+ +-C2B-----+ +-C2D-----+ +-C2F-----+ +-C2I-----+ +-C2P-----+-----+ +-----(scale)---+ +-C2V-----+ +-C2X-----+ +-B2C-----+ +-D2C-----+ +-F2C-----+ +-I2C-----+ +-P2C-----+-----+ +-----(scale)---+ +-V2C-----+ +-X2C-----+ +-f2t-----+</pre>
	Output:
	<pre> ---+-----+-----+ (1) .n. +-+--NEXTWord--+-----+ (1) .n. +-+--NWord-----+ +-+--columnrange-----+ (1) No blanks allowed</pre>
	Alignment:
	<pre> ---+-----+ +-Left---+ +-Center---+ +-Right---+</pre>
spill	Not implemented. Word wrap.

split	>>>SPLIT--+-----+---+ +--ANYCase--+ +--MINimum--number--> +--AT-----+ +--BLANK-----+ >-+-----+---+---+---+ +---+---+---BEFORE--+ +---NOT--+ +--- target ---+---+---+ +--snumber-+ +--AFTER---+ +--number---+ target: ---+---xrange-----+---+ +---+---STRING---+/string/-+ +---ANYof---+
sql	Not implemented. OS dependent. See sqlselect for Pipes for NetRexx extension.
sqlcodes	Not implemented. OS dependent.
sqlselect	Pipes for NetRexx only. Extension to SQL uses jdbc to select from any jdbc en
stack	Not implemented. OS dependent.
Starmonitor	Not implemented. OS dependent.
starmsg	Not implemented. OS dependent.
starsys	Not implemented. OS dependent.
state	Not implemented. Determine if file(s) exist.
statew	Not implemented. Determine if file(s) exist.
stem	>>>STEM--stem-----><
stema	Pipes for NetRexx only. Append to an existing stem, exits if the argument is
stemr	Pipes for NetRexx only.
stemw	Pipes for NetRexx only.
storage	Not implemented.
strasmfind	Not implemented. Assembler.
strasmnfind	Not implemented. Assembler.
strfind	>>>---STRFIND---+-----+---/string/-+ +---ANYcase---+><
strfrlable	>>>---STRFRLABEL---+-----+---/string/-+ +---ANYcase---+><
strip	Not implemented. Remove leading / trailing characters.
strliteral	>>>STRLITERAL---+-----+ +---APPEND---+ +---/string/-+><
strnfind	>>>STRNFIND---+-----+---/string/-+ +---ANYcase---+><
strtolabel	>>>STRTOLABEL---+-----+---/string/-+ +---ANYcase---+><
strwhileable	Not implemented. Select consecutive records.
subcom	Not implemented. OS dependent.
synchronize	Not implemented. Synchronizes a multistream pipeline.
take	+--FIRST---+ +--1---+ >>>TAKE---+-----+---+ +--LAST----+ +--n---+ +--*---+
tape	Not implemented. Old equipment.
timer	

tcpclient	<p>Simple tcpclient implementation. The options implemented are similar to the C</p> <p>linger - wait a bit before terminating the last read (units SECONDS)</p> <p>timeout - wait this long before timing reads out (units MS)</p> <p>deblock - If deblock is omitted a copy stage is used.</p> <p>group - similar to CMS. A delimited string containing a stage is expected. You can use a run of stages, but its is dangerous since you to know the stage sep character being used...</p> <p>greeting - expect a greeting message and discard it</p> <p>nodelay - use the nodelay option</p> <p>keepalive - enable keep alive socket option</p> <p>oneresponse - synchronize cmd/replies</p>
tcpdata	<p>Simple tcpdata implementation.</p> <p>linger - wait a bit before terminating the last read (units SECONDS)</p> <p>timeout - wait this long before timing reads out (units MS)</p> <p>deblock - If deblock is omitted a copy stage is used.</p> <p>group - similar to cms. A delimited string containing a stage is expected. You can use a run of stages, but its is dangerous since you need to know the stage sep character being used...</p> <p>nodelay - use the nodelay option</p> <p>oneresponse - synchronize requests/replies</p>
tcplisten	<p>Simple tcplisten implementation. You can only supply the port and a timeout value, which is ignored unless tcplisten's output stream has been severed, in which case tcplisten terminates.</p> <p>If input stream 0 is connected, tcplisten does a peekto before calling the accept method. The object is consumed after the output of the socket object returns.</p>
timestamp	<pre>+--8--+ >>---TIMEStamp---+-----+----->< +--n--+</pre>
timer	Pipes for NetRexx only. Time pipes
tokenise	<pre>>>---+---TOKENISE---+/string/---+----->< +---TOKENIZE---+ +---/string/---+</pre>
tolabel	<pre>>>---TOLABel---+-----+----->< +--string--+</pre>
totarget	<pre>>>---TOTARGET---stage---+-----+----->< +--operands---+</pre>
udp	Not implemented. Read / Write TCP records.


```
zone      >>--ZONE---+-----+-->
          +---+--WORDSEParator---+---+--char---+---+
          +-+--WS-----+     +-+--hexchar---+
          +-+--FIELDSEparator---+     +-+--BLANK---+
          +-+--FS-----+     +-+--SPACE---+
          >-+--Words----wnumberrange---+---->
          +-+--Fields---fnumberrange---+
          +-+--cnumberrange-----+
          >-+-----+---+-----+---+stage---+-----+--><
          +-+--CASEI--+     +-+--REVERSE--+           +-+--operands--+
```

Last modified: Thu Sep 5 21:30:14 AST 2019

Appendix A

- .50 - Released May 30, 1999
 - Fixed a stall occurring when interrupted threads, with the interrupt caught by ThreadPool, were reused.
 - Fixed a thread safety problem in ELASTIC
 - Improved the timeout options in TCPDATA and TCPCLIENT, they also byte[] instead of strings. This was done since converting to and from strings sometimes scrambles binary data (more research on encodings...)
 - Changed DELBLOCK it now handles byte[] to help keep tcpdata and tcpclient efficient. The EOF option was broken, its fixed now.
 - Changed DISKR, DISKW and DISKA to handle byte[] when using streams.
 - Added INSERT which handles byte[]. This should be used instead of SPECS to add LF or CR .
 - Changes SERIALIZED to use byte[].
- 0.49 - Released May 21, 1999
 - compiled with 1.2.1 and NetRexx 1.148
 - Added preliminary support added to .njp compiler for files containing java source! See the (some what messy) java samples in vectort1.njp, overtest.njp and addtest4.njp
 - Added code to generate a dummy .nr file containing the public class in a .java file. This allows NetRexx to compile class that depend on the java source.
 - Modified sort to accept arguments in the same order as CMS
 - Fixed rc logic in drop stage
 - Fixed shortcut code for {n} where n is numeric.
- 0.48 - Released May 16, 1999
 - Fixed a (nasty) bug involving reusing pipe objects.
 - Added the reuse() method to the stage class. To use it override it in your stage. It was added so there was a foolproof way to reset a stage when its pipe object is reused. (doSetup is intended for use with dynamic arguments in call or added pipes)
 - Added the cont option and defaulted it to comma.
 - fixed return code logic in some stages and in selectInput/Output
 - Added the Emsg methods
 - Added argument debug option (128)
 - There are no more final methods
 - Much improved error reporting from stages via new Emsg method
- 0.47 - Released Jan 3, 1999

- recompiled with 1.1.7A and netrexx 1.148
 - UNIQUE repaired?
 - Added stages to access java objects easily
 VECTOR, VECTORR, VECTORW, VECTORA for java.util.vector
 ARRAY, ARRAYR, ARRAYW, ARRAYA for Object[]
 HASH, HASHR, HASHW, HASHA for java.util.Hashtable
 DICT, DICTR, DICTW, DICTA for java.util.Dictionary
 The hash stages mostly map directly to DICT stages. The exception
 is HASHW which uses the clear() method of Hashtable.
 - Modified LITERAL to be able to put any object into a pipe
 - Modified pipe package to store arguments in a hashtable instead of
 a rexx stem - arguments can now be of any class. Use the arg(null)
 method to get an object argument.
- 0.46a - Released Oct 14, 1998
- recompiled with 1.1.7
 - TCPLISTEN now supports an input stream to be used to pace accepts
- 0.46 - Released Sept 20, 1998
- COMMAND, CHANGE, FILE, LOCATE, DROP, LOOKUP, TCPCLIENT, TCPLISTEN
 SQLSELECT, CONSOLE, TCPDATA, NOEOFBACK improved.
 - Jeff improved the testing process with the addition of the COMPARE
 stage, he also upgraded many of the tests.
 - Added the buildtests pipe, it builds a test script to be run with:
 test > output < console.data
 - Unexpected exceptions should no longer hang pipes
- 0.45 - Released Sept 9, 1998
- * Recompile all your stages. To fix a commit problem I had to
 change the _stage interface class...
 - tcpclient restart problems with oneresp active fixed.
 - commit now returns the current return code of the pipe.
 - fixed minor errors in tcpclient and disk.a
- 0.44 - Released Sept 8, 1998
- * a recompile of pipes using STEM is required
 - smart DISK, FILE and STEM stages now exist.
 - Made to and from synonyms for in and out in REXX and STRING stages.
 - Added stream option to DISKR and DISKW to read raw streams.
 - Added DISKSLOW and SERIALIZED stages.
 - Now DISK, DISKR, DISKW, DISKA and DISKSLOW have FILE synonyms.
 - Deadlock detection improvements.
 - TCPDATA & TCPCLIENT optimized once again.
 - selectAnyInput could deadlock - fixed.
 - interrupting a pipe now kills it - use this with care (ie. kill -9)
 - Pseudo methods njpRC() and njpObject() are recognized by the pipes
 compiler and return the pipe's RC or object respectively.
- 0.43 - Released August 30, 1998
- Fixed deadlock detection to see commit deadlocks.
 - Added rest of code to handle improved StageError logic.
 - Added stage templates (template*.nrx) in the njpipes directory.

- Added a debug flag (64) to trace all StageError rased by the stage class.
- 0.42 - Not released
- * A recompile of pipes using TCPCLIENT, TCPDATA is required.
 - * A recompile of pipes using REXX, STRING, ZONE, CASEI is recommended.
 - Updated the comments in _stage to reflect the possible StageError and return codes that can be issued.
 - Added the DEBLOCK stage and reworked TCPDATA, TCPCLIENT & GATE.
 - Improved eofReport processing and added a new option 'either' that will trigger a StageError when any stream, input or output, severs.
 - Fixed variable subsitution so multiple variables passed to a stage will work.
 - Added the ability to pass thru arguements to callpipe and addpipe.
 - Fixed a problem with some StageExits requiring stage_reset methods.
 - Added a function to utils to help assign smarter name to classes generated by StageExits.
 - Added counter method to stage. use to count external waits so deadlock/stall detection is not fooled.
- 0.41 - Released August 23, 1998
- * removed OBJ2REXX, OBJ2STRING stages, use REXX and STRING stage modifiers.
 - * pipes using TCPDATA, TCPCLIENT & LOOKUP should be recompiled
 - enhanced REXX stage modifier via an object2rexx improvement in pipes/utils.nrx
 - optimized ThreadPool startup times. No setName and only use setPriority when its required.
 - made it possible to optimized stage startup time when arguements are static. See TCPDATA, TCPCLIENT & LOOKUP
 - faq.txt enhanced
- 0.40 - Released August 14, 1998
- * All pipes MUST be recompiled. Old pipe class files will stall.
 - OBJ2REXX is depreciated and will be removed, use the REXX stage.
 - added REXX and STRING stages to convert objects entering and leaving a stage to rexx or string. Inorder to avoid nasty class conflicts, REXX and STRING are implemented in _rexx and _string. The compler adds the '_' when necessary (any stage can use this feature).
 - fixed an intermitant stall in callpipe (was completing too fast :-)
 - fixed a stall occurring between shortStreams and COMMAND
 - optimized pipe startup time in pipe.class and via the compiler.
 - optimized rc, commit, deadlock, threadpool code
- 0.39 - Released August 9, 1998
- WAIT_COMMIT and WAIT_ANY are now used in the call/addpipe logic
 - callpipe was not notifiing its pipe when ending leading to an very intermitant hang.
- 0.38 - Released August 3, 1998
- * All your stages must be recompiled. Recompile your pipes to exploit the pipe & thread pool performance improvements.

- fixed and optimized commit logic.
 - implement a pool for pipes to decrease overhead.
 - implement a pool for threads to decrease overhead.
 - compiler fix to propagate return codes from stageExits (thanks Jeff).
 - signal StageError('... in all stageExits modified to
signal StageError(13,'Error - 'pInfo' - ...
 - UNIQUE stage added by Jeff. It exploits stageExit.
 - COMMAND stage was not starting its threads correctly.
 - SORTs in different pipes could corrupt each other. Thanks René,
- 0.37 - Released July 25, 1998
- * A recompile of pipes using SORT is required
 - added NOEOFBACK, TOTARGET and FRTARGET.
 - removed a protected method from dump(), added arg() to the dump
 - upgraded SORT, sortRexx to exploit IRange and stageExit, optimized use, and factored the sort algorithm out of sort/sortRexx.
 - multiple sort stages no longer try to share static variables...
 - the compiler just uses the stage name (not args) when naming stages
- 0.36 - Released July 19, 1998
- * A recompile of ALL pipes with stages using IRANGE is required.
(CHANGE, DEAL, JOINCONT, LOCATE, LOOKUP, PICK, XLATE & ZONE)
 - * pipes using NFIND, NLOCATE, STRNFind or SORT also need to be recompiled
 - Added BuildIRangeExit and other methods to an updated IRange class. Using 'zone range stage ...' will be faster than 'stage range ...' when the range consists of n.c or n-c (s).
 - NFIND, NLOCATE, STRNFind implemented via stageExit and NOT
 - Fixed bugs in, JUXTAPOSE, FIND, STRFIND, SORT, COMMAND, CHANGE
 - The compiler was not calling stageExit in the correct order when several calls were needed to build the stage. (zone w1 nfind..)
- 0.35 - Released July 16, 1998
- Jeff Hennick pointed out a buglet that effected LOOKUP, ZONE and PICK that could occur with complex ranges, I found another bug in strliteral
 - Jeff Hennick updated this doc with information on IRange and DString
- 0.35 - Released July 15, 1998
- * A recompile of ALL pipes using ZONE, TCPCLIENT, TCPDATA, PREFIX and APPEND is required.
 - prefix and append can now be labeled, tcpclient and tcpdata now use a stage, instead of a pipe, to group data.
 - added compiler support for negative stream numbers. This is intended to be used by stageExit. See append, prefix, tcpdata and tcpclient.
 - Redefined rexxArg() and stageArg() to simplify the compiler.
 - selection stages are no longer defined as final.
 - SelectInput(0) and selectOutput(0) are always called by the stage implementation so they can be overridden...
 - Reimplemented ZONE using stageExit, added CASEI using the same

- technique. In theory NOT could be done the same way but, to avoid some recursion problems NOT is staying in the compiler.
- StageExit modified to allow it to pass back another stage to call. see ZONE, CASEI and NOT.
- 0.34 - Released July 11, 1998
- minor reportEOF(any) logic fix
 - improved command stage, threads used to process stdout and stderr. added zone, pad, lookup, pick, upgraded juxtapose, fixed bugs in specs & buffer.
 - added pad option to setIRange method
- 0.33 - Released July 5, 1998
- added rexxArg() and stageArg() methods to utils.nrx for use by the compiler to query stages about what they expect their arguments to contain. This allowed the compiler to be simplified.
- \$ - locate now handles null arguments correctly. literals now include leading blanks. Thanks for pointing out the problem René.
- \$ - René Jansen contributed the timestamp stage.
- logic modified to stop output() from getting an EOF when the output object has been peeked. The peek status is also displayed by the dump() method and hence by deadlocks.
 - minor specs bug fixes (next.n and nextw.n output specs now work)
 - modified the compiler to invoke stageExit(rexx,rexx) method. This allows stages to generate code and/or change the pipe topology. See specs, append, prefix, change and xnop, in the stages directory.
 - modified StageError in preparation for usage changes.
 - removed the Range class - Jeff's code is better and anything that could be done with Range can be done using stageExit.
 - Jeff fixed bugs in change and join and added:

fblock	joincont	notinside	outside
inside			
- 0.32 - Released June 20, 1998
- Jeff updated these stages adding a few new ones too:
- | | | | |
|---------|----------|---------|--------|
| abbrev | between | split | locate |
| nlocate | strnfind | strfind | nfind |
| find | chop | | |
- minor documentation updates
 - the Range class is deprecated and will be removed. Use the replacements Jeff created (see pipes\utils.nrx and stages\).
- 0.31 - Released June 17, 1998
- modified count, drop, take and deal to handle non rexx objects when possible
- 0.31 - Released June 16, 1998
- improved eofReport(ANY) logic to trigger when waiting on output and a different output stream servers.
 - factored the source for utils.class out of stages so there is a class to add (probably static) shared methods for all stages
 - fixed a deadlock that occurred between shortStreams and exit

- (severInput)
- Jeff Hennick updated many stages to work at CMS or near CMS levels.
- | | | | | |
|---------|---------|---------|------------|-------|
| append | deal | join | strfrlabel | xlate |
| buffer | drop | literal | strliteral | |
| change | fanin | locate | strtolabel | |
| console | fanout | split | take | |
| count | frlabel | strfind | tokenize | |
- All of Jeff's changes are GNUed. See CopyLeft.txt in the njpipes directory.
- 0.30 - Released May 24, 1998
- fixed logic in core classes to post all pending servers and not clear them too early either, this corrects a problem seen on Multiprocessor machines.
- 0.29 - www page update (documentation) May 20
- deadlock section updated
 - installation verification example corrected!
- 0.29 - Released May 17, 1998
- added obj2rexx stage, tolable stage courtesy of Chuck Moore.
 - enhanced change to support a single range
 - Added setJITCache(Hashtable) method to pipes. This can be used to build a global object cache in programs calling pipes. The name of the Hashtable is passed to pipe/callpipe/addpipe via a cache parameter.
 - Added support for reportEOF options. This support is not too well tested - some good testcases are needed.
- 0.28 - Released May 9, 1998
- Enhanced parsing in specs (word2.1 would work, word 2.1 would not)
 - Fixed COPY for a NT jit bug, fixed locate so NOT LOCATE would work, updated LITERAL not to use more than one exit(rc)
 - Fixed a compiler problem that would hit multistreamed pipes using append or prefix.
 - Any options not consumed by njp are passed on to nrc and java. Mainly for use from the command line, use with care in .njp files...
 - Fixed shortStreams() so it works correctly when shorting streams in a stage with multiple streams.
 - Tested all 8 addpipe forms and fixed runtime to work with all test cases
 - modified filternjp to accept *in and *out without additional labels
 - reenabled stop() in exit code...
 - added gate, dam, tokenize, juxtapose and courtesy of Chuck Moore, frlabel stages
- 0.27 - Released May 3, 1998
- Automated the generation of in/outStream calls. For this to work the labels need to be of the form *in0: or *out0: where the '0' is replaced by the input or output stream to connect to.
 - Fixed compiler/filter problems with stems

- Tightened range checking code in specs, fixed problem with delimited ranges. Specs was compiling the NetRexx EXIT command...
 - Fixed a problem where output was not seen that objects were consumed when using sipping pipes...
 - Fixed a problem where severing an output stream did not cause the stages stacked on the node's outlist to see the sever
 - Fixed a problem where the stage issuing a callpipe was not seeing the called pipe end
 - Added debug option to pipes compiler
 - Repaired commit and added commit levels to dump() method
 - Fixed problems with callpipe serving several outputs, unstacking the saved stream was selecting it...
 - Modified tcpclient and tcpdata to use a secondary thread to receive the tcpip inputs.
 - Now keep a referenced object for each pipe/stage so the JIT does not throw away its work and call/addpipes in loops work faster.
 - in/outStreamState now return -1 when autocommit is enabled and the stream is unused.
- 0.26 - Released April 26, 1998
- Added selection methods to compiler (see getRange in section 4 and the locate stage an example#)
 - Added the specs stage. The compiler builds a stage to process the specs, reducing overhead.
 - Added tcp/ip stages
 - Fixed problems with servers using addpipe
- 0.25 - Optimized some stages using jinsight from www.alphaworks.ibm.com. This more than doubled the speed of some stages.
- fixed bugs in fanin, diskw
 - Added netrexx filters to extract pipes, extended the functions of .njp files (multiple pipes in a file and .njp files can now contain netrexx code with pipe/callpipe/addpipe)
 - fixed a timing bug in deadlock detection.
 - xlate and sqlselect stages contributed by René Jansen added
- 0.24 - Release Feb 98
- modified the compiler so the syntax of pipes from the command line is the same as pipes from .njp files
 - added the sort stage, the sortClass interface and the sortRexx example implementation
 - added the timer stage
- 0.23 - fixed minor compiler errors (20 Dec 97)
- not stage modifier added.
 - errors in this page corrected, NT install information added.
 - modified diskr/diskw to use Buffered Streams.
- 0.22 - second public release
- 0.21 - enabled auto commit, stages start at a commit level of -2 and commit to a level of -1 at the first readto, peekto or output. nocommit disables the auto commit. This feature has not been

- completely tested (yet).
- fixed compiler not to call netrexx if one of its pipes deadlocks
- 0.20 - Upgraded to May version of the NetRexx compiler (Thanks Mike!)
this changed the compiler interface. NetRexx from May 10 or
later is now required.
- nocommit added to _stages, though its a nop for now
 - modified the compiler class to use the May 10th NetRexx compiler
- 0.19 - initial public release (4 May 97)

List of Figures

1	Run in the NetREXX Workspace	5
2	Run from the OS command line	5
3	Precompile a Pipeline from the OS command line	6
4	example 1	8
5	example 2	9
6	example 3	10
7	example 4	11
8	example 5	11
9	BAGVENDT under VM/CMS	13
10	bagvendt.nrx under NetREXX	13
11	Deadlock detection	21
12	TCP/IP Client/Server compile	28
13	TCP/IP server	28
14	TCP/IP requestor	28

List of Tables

Listings

Example Listing	iii
---------------------------	-----

Index

Rexx, 12, 24, 26
arg, 24, 25
binary, 27
catch, 7, 12, 25-27
class, 6, 12, 24, 25, 27
do, 6, 12, 25
end, 6, 7, 12, 24-27
exit, 7, 12, 25, 27
extends, 6, 12, 24, 25, 27
final, 6, 24, 25
forever, 6, 12, 25-27
import, 6, 12, 25, 27
loop, 6, 12, 24-27
method, 6, 12, 24, 25, 27
options, 27
over, 24
public, 24, 25, 27
rexx, 6, 27
say, iii, 24
signal, 12
static, 24
to, 24, 27
where, 29

ISBN 978-90-819090-3-7

A standard linear barcode representing the ISBN number 978-90-819090-3-7.

9 789081 909037 >