Pipelines Reference

Ed Tomlinson **Jeff Hennick** **Rene Jansen**

Version 3.04 RC1 of May 18, 2015

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

Publication Data

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

Contents

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 Reference	The Data Flow oriented companion to NetRexx, with its CMS Pipes 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:

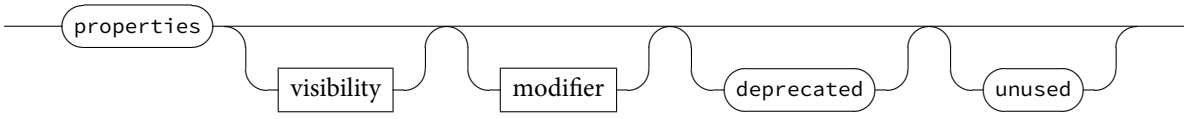
- Body text is in this font
- Examples of language statements are in a **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are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



1

Introduction

A Pipeline, or Hartmann Pipeline[1], is a concept that extends and improves pipes as they are known from Unix and other operating systems. The name pipe indicates an interprocess 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 ported to TSO to run under MVS 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.[2] It is written in NetRexx and pipes and stages can be defined using this language. The resulting code 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.

[1] `http://en.wikipedia.org/wiki/Hartmann_pipeline`

[2] And indeed, it can also be used from programs written in the Java Language.

2

The Pipeline Concept

2.1 What is a pipeline?

The *pipeline* terminology is a metaphore derived from plumbing. Fitting two or more pipe segments together yield 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 exclamation mark (!) is used as *stage separator*, while the solid vertical bar | can be used as an option when specifiying the local option for the pipe, after the pipe name.

When looking a two adjaced 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.

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 `diskr` for diskread and `diskw` for diskwrite; 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. Figure X 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 throug a device independent standard interface.

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.

2.4 Building the pipeline

Until now everything was just theory, but now we are going to show how to compile and run a pipeline. The executable script `pipe` is included in the NetRexx distribution to specify a pipeline and to compile NetRexx source that contains pipelines. Pipelines can be specified on the command line or in a file, but will always be compiled to a .class file for execution in the JVM.

Listing 2.1: Hello World

```
1 pipe '(hello) literal "hello world" ! console'
```

This specifies a pipeline consisting of a source stage `literal` that puts a string ("hello world") into the pipeline, and a `console` sink, that puts the string on the screen. The pipe compiler will echo the source of the pipe to the screen - or issue messages when something was mistyped. The name of the classfile is the name of the pipe, here specified between parentheses. Options also go there.

We call execute the pipe by typing:

```
java hello
```

Now we have shown the obligatory example, we can make it more interesting by adding a `reverse` stage in between:

Listing 2.2: Hello World2

```
1 pipe '(hello) literal "hello world" ! reverse ! console'
```

When this is executed, it dutifully types "dlrow olleh".

If we replace the string after `literal` with `arg()`, we then can start the `hello` pipeline with a an argument to reverse:

Listing 2.3: Hello World3

```
pipe "(hello) literal arg() ! reverse ! console"
```

and we run it with:

```
java hello a man a plan a canal panama
```

and it will respond:

```
amanap lanac a nalp a nam a
```

which goes to show that without ignoring space no palindrome is very convincing - which we can remedy with a change to the pipeline: use the `change` stage to take out the spaces:

Listing 2.4: Hello World4

```
pipe "(hello) literal arg() ! change /" "// ! console"
```

3

Device Drivers

A device driver reads from a device (for instance a file on disk, on the network, or from the shell) or writes to a device; in some cases it can both read from and write to the device. An example of a device driver is `disk` which reads and writes files. You can use it on the left-hand side of the pipeline as an input device and on the right-hand side of the pipeline as an output device

4

Filters

5

Record Selection

6

NetRexx Pipelines Implementation

NetRexx Pipelines enables us to follow the usage model of CMS Pipelines closely; in fact, the documentation for the mainframe product can be used for most stages.

6.1 Installation and verification

To run NetRexx Pipelines a running NetRexx installation is needed. To write your own pipes or stages you need compilers for both Java and NetRexx. The core classes for pipes and stages are in the archive NetRexxF.jar. This file may be used on the -cp option or added to your CLASSPATH, as indicated in the *NetRexx Quickstart Guide*.

To test the installation, we can run a pipeline from the command line. Running a pipeline from the command line To run a pipeline from the commandline, type:

Listing 6.1: Count Words

```
1 pipe (test) 'literal arg() ! dup 999 ! count words ! console'
```

The first time you use the pipe command in a new directory it will create a default pipes.cnf file for you. When the `pipe` command is not on your path, you can also use:

Listing 6.2: Test Dup

```
1 java org.netrexx.njpipes.pipes.compiler (test) 'literal arg() ! dup 999 ! count words
    ! console'
```

You should see a message that the pipe compiler is processing your pipe and soon after that messages from the NetRexx compiler as it processes the pipe. To run the pipe type:

```
java test some words
```

The pipe should then output:

```
 2000
```

7

Differences with respect to the CMS Pipelines version

8

Developing stages in NetRexx

Writing your own pipes or stage is simple. Take a look at the source of the supplied stages in the stages directory. Here are some more examples. The first shows how to use a pipe in a NetRexx program: – examples/testpipe.njp – to compile use: pipe testpipe.njp – or: java njp testpipe.njp – to execute use: java testpipe some text

```
class testpipe

  method testpipe(avar=Rexx)

    F = Rexx 'abase'
    T = Rexx 1

    F[0]=5
    F[1]=222
    F[2]=3333
    F[3]=1111
    F[4]=55
    F[5]=444

    pipe (apipe stall 1000 )
      stem F ! sort ! prefix literal {avar} ! console ! stem T

    loop i=1 to T[0]
      say 'T['i']='T[i]
    end

  method main(a=String[]) static

    testpipe(Rexx(a))
```

A couple things can be inferred from this example. First its simple to pass rexx variables to pipes using STEM. Also look at the phrase avar. It passes the Rexx 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 stagesep 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 or java code you can use arg('name') to get the named argument. If the class of the arguement 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.

```
  -- over.nrx
class over extends stage final

  method run() public
     a = getRexx(arg())
    loop i over a
```

```
      output(a[i])
    catch StageError
      rc = rc()
    end

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
```

```
import org.netrexx.njpipes.pipes.

class calltest extends stage final

method run() public

  do
    a = arg()

    callpipe (cp1) gen {a} ! *out0:

    loop forever
      line = peekto()
      output(line)
      readto()
    end

  catch StageError
    rc = rc()
  end

exit(rc*(rc<>12))
```

Running the callt1 pipe with an argument of 10 would pass the 10 to calltest via and 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.

```
-- similar to examples\addtest.njp

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 classCastExceptions. 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.

To get a good idea of what can be done with pipes look at the tasktest pipe in the examples directory. It, using code from Melinda Varins 'Cramming for the Journeyman Plumber Exam' paper, implements the shell of a multitasking server - using about eight stages. The file examples/tcptask.njp contains an example of this technique being used.

9

Deadlocks

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

```
java njp (deadlock) literal test ! a: fanin ! console ! a:
Pipes for NetRexx and Java version 0.33
Copyright (c) E. J. Tomlinson , 1998.  All rights reserved.
Building pipe deadlock
NetRexx portable processor, version 1.140
Copyright (c) IBM Corporation, 1998.  All rights reserved.
Program deadlock.nrx
Compilation of 'deadlock.nrx' successful

njpipes/examples]java -nojit deadlock
test
Deadlocked in deadlock

Dumping deadlock Monitored by deadlock

 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
 -> out 0 fanin_2 1 test

 fanin_2
 Running rc=0 commit=-1 Flag=101
 -> 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=101
 -> in  0 fanin\_2 1 test
 -> out 0 fanin\_2 0 test

Dumped Pipe deadlock Flag 40F rc=16
```

```
RC=16
```

We can see that there are three stages Running. 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 datawaiting on in stream 1. Hence the stall. 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 syncronized blocks. When a stage has a pending sever or autocommit flag bits are set too.

10

The Pipe Compiler

11

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 njpipesC.jar file; also, the source of these stages is delivered in the package. General notes on the built-in stages:

1. The underlying technology, the JVM, and the chosen implementation language, NetRexx, cause the character representation to be Unicode.
2. Most of the stages expect the objects in the pipeline to be of type Rexx

12

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 SERIALIZE 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 .nrx 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 arguements 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 arguements 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 arguement 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 acess 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 arguements in a hashtable instead of
     a rexx stem - arguements can now be of any class.  Use the arg(null)
        method to get an object arguement.
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 diska.
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 synonoms for in and out in REXX and STRING stages.
      - Added stream option to DISKR and DISKW to read raw streams.
      - Added DISKSLOW and SERIALIZE 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 reconized by the pipes
        compiler and return the pipe's RC or object respectivily.
0.43  - Released August 30, 1998
      - Fixed deadlock dection 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 rasied 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

- exhanced 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 occuring 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 proprogate 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 bugglet 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 negitive 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 hense 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 docuementation updates
          - the Range class is depreciated 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 severs.
          - 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 occured 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 severs and not clear them too early either, this corrects a problem seen on Multiprocessor machines.

0.29 - www page update (docuemention) May 20
- deadlock section updated
- installation verification example corrected!

0.29 - Released May 17, 1998
- added obj2rexx stage, tolabel 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 Hasttable 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 stema

- Tighted range checking code in specs, fixed problem with delimited ranges. Specs was compiling the NetRexx EXIT command...
- Fixed a problem where output was not see 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 servering several outputs, unstacking the saved stream was selecting it...
- Modified tcpclient and tcpdata to use a secondary thread to recieve 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 severs 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

List of Tables

Listings

Index

ISBN 978-90-819090-3-7