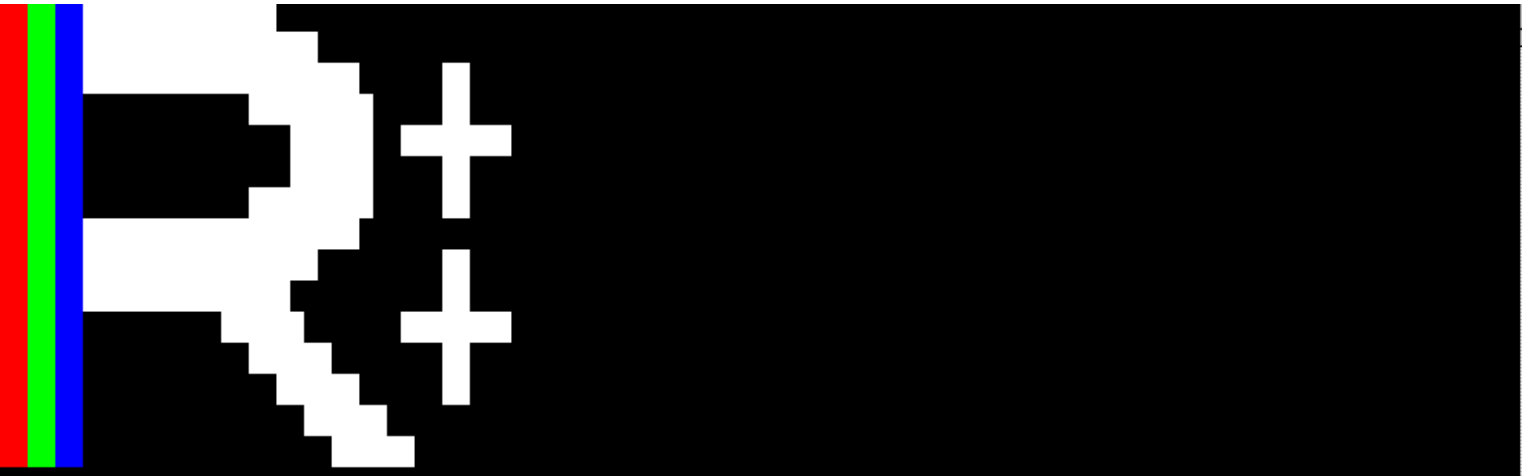
RPL++ 1.5.0A
Reference Manual

1.5.0A



RPL++ 1.5.0A
Reference Manual

Preface

This manual contains specific information about the RPL++ 1.5.0A.

Contents

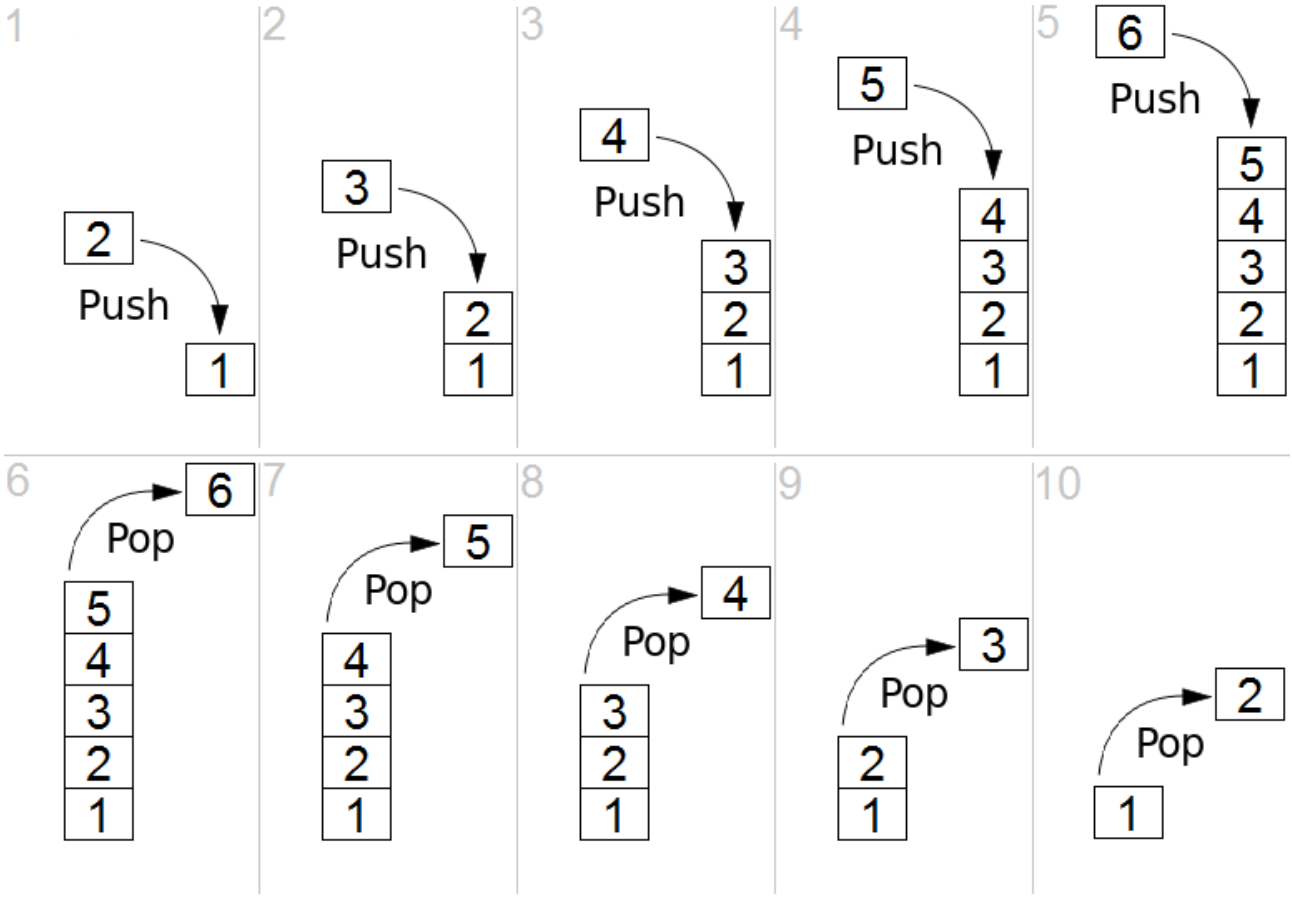
Chapter 1. Syntax

Basic

RPL++ is reverse polish notation programming language.
So, 1 + 1 in RPL++ is:

```
1 1 +
```

And, RPL++ is stack-based.



Types

RPL++ has only 6 types.

- String – `"string"`

  ## Escape sequences

  - Use “\xXX” for U+00XX
  - Use “\uXXXX” for U+XXXX
  - Use “\r” for CR (Carriage Return)
  - Use “\n” for LF (Line Feed)
  - Use “\\” to escape backslash
- Number - `123`
- undef
- notnum
- nil
- Array (See Chapter 5)

Comment

There are 2 comment types, single-line comment, and multi-line comment.

- Single-line comment - // Comment
- Multi-line comment - /* Comment */

Chapter 2. Mathemetical Operators

Addition (+)

Adds 2 numbers.
Usage: `n1 n2 +`
Example: `1 1 + .NL`
Result: `2\n`

Subtraction (-)

Subtracts n1 from n2.
Usage: `n1 n2 -`
Example: `1 2 - .NL`
Result: `-1\n`

Multiplication (*)

Multiplies n1 by n2.
Usage: `n1 n2 *`
Example: `2 3 * .NL`
Result: `6\n`

Division (/)

Divides n1 by n2.
Usage: `n1 n2 /`
Example: `1 2 / .NL`
Result: `0.5\n`

Remainder (%)

Calculates the remainder when n1 is divided by n2.
Usage: `n1 n2 %`
Example: `3 2 % .NL`
Result: `1\n`

Misc (_)

If mode is 0: Calculates floor(n1).
If mode is 1: Calculates ceil(n1).
If mode is 2: Calculates round(n1).
If mode is 3: Calculates n1 to the power of n2.
Usage: `n1 [required when mode is 3: n2] mode _`
Example: `2 3 3 _ .NL`
Result: `8\n`

Random (?)

Generates a random number between 0 and 1.
Usage: `?`
Example: `? .NL`
Result: `0.39805341716781983\n`

Factorial (!)

Calculates the factorial of n.
Usage: `n !`
Example: `5 ! .NL`
Result: `120\n`

Chapter 3. Input-Output Operators

Stdout output (.)

Prints x.
Usage: `x .`
Example: `1 .`
Result: `1`

Stdout output with newline (.NL)

Prints x with newline.
Usage: `x .NL`
Example: `1 .NL`
Result: `1\n`

Stdin input (.?)

Gets the input from stdin by line.
Usage: `.?`
Example: `.? .NL`
Input: test
Result: `test\n`

Stdin raw input (,?*word*)

Gets the input from stdin and sends the data to the word.
Usage: `,?`*`word`*
Example: `wd-begin . wd-end input ,?input`
Input: [Ctrl+C]
Result: `\u0003`

Newline (`NL`)

Prints newline.
Usage: `NL`
Example: `NL`
Result: `\n`

File input (#>)

Reads the file.
Usage: `filename #>`
test.txt: Hello!
Example: `"test.txt" #> .NL`
Result: `Hello!\n`

File output (#<)

Reads the file.
Usage: `data filename #>`
Example: `"Hello!" "test.txt" #<`
test.txt: Hello!

Checks if the file exists (#?)

Checks if the file exists.
Usage: `filename #?`
test.txt: Hello!
Example: `"test.txt" #?`
Result: `1\n`

Chapter 4. Bitwise & Logical Operators

Logical AND (&) / Bitwise AND (& .)

Calculates b1 AND b2.
Usage: `b1 b2 &`
Example: `1 0 & .NL`
Result: `0\n`

Logical OR (|) / Bitwise OR (| .)

Calculates b1 OR b2.
Usage: `b1 b2 |`
Example: `1 0 | .NL`
Result: `1\n`

Logical XOR (^) / Bitwise XOR (^ .)

Calculates b1 XOR b2.
Usage: `b1 b2 ^`
Example: `1 0 ^ .NL`
Result: `1\n`

Logical NOT (~) / Bitwise NOT (~ .)

Calculates NOT b.
Usage: `b ~`
Example: `1 ~ .NL`
Result: `0\n`

Chapter 5. Array Operators

Creates an array (])

Crerates an array with length len.
Usage: `e0 e1 e2 … e(len-1) len ]`
Example: `1 2 3 3 ] .NL`
Result: `<Array [1, 2, 3]>\n`

Extract ([)

Extracts the array to the stack.
Usage: `array [`
Example: `1 2 3 3 ] [ .NL`
Result: `3\n`

Gets the length (][)

Gets the length of the array.
Usage: `array ][`
Example: `1 2 3 3 ] ][ .NL`
Result: `3\n`

Gets the value (@)

Gets the value of the array.
Usage: `array index @`
Example: `1 2 3 3 ] 1 @ .NL`
Result: `2\n`

Compress (#&)

Filter the array.
Usage: `pattern array #&`
Example: `1 0 1 3 ] 1 2 3 3 ] #& .NL`
Result: `<Array [1, 3]>\n`

Range (. . .)

Creates a range array.
Usage: `start end ...`
Example: `1 4 ... .NL`
Result: `<Array [1, 2, 3]>\n`

Zip ([+])

Zips the array to the another array.
Usage: `a1 a2 [+]`
Example: `1 2 3 3 ] 4 5 6 3 ] [+] .NL`
Result: `<Array [<Array [1, 4]>, <Array [2, 5]>, <Array [3, 6]>]>\n`

Unzip ([-])

Unzips the zipped array.
Usage: `array ...`
Example: `1 4 2 ] 2 5 2 ] 3 6 2 ] 3 ] [-] . .`
Result: `<Array [4, 5, 6]><Array [1, 2, 3]>`

Sets the value ([$]<)

Sets the value of the array.
Usage: `array index value [$]<`
Example: `1 2 4 3 ] 2 3 [$]< .NL`
Result: `<Array [1, 2, 3]>\n`

Pop ([$]^)

Removes the top of the value of the array.
Usage: `array [$]^`
Example: `1 2 3 3 ] [$]^ .NL`
Result: `<Array [1, 2]>\n`

Zero-filled array ([$] -)

Creates a zero-filled array.
Usage: `length [$]-`
Example: `3 [$]- .NL`
Result: `<Array [0, 0, 0]>\n`

Array that is filled with the value ([$] *)

Creates an array that is filled with the specified value.
Usage: `value length [$]*`
Example: `3 5 [$]* .NL`
Result: `<Array [<Array [3]>, <Array [3]>, <Array [3]>, <Array [3]>, <Array [3]>]>\n`

Concatenation ([$] +)

Concat the array to another array.
Usage: `a1 a2 [$]+`
Example: `1 2 3 3 ] 4 5 6 3 ] [$]+ .NL`
Result: `<Array [1, 2, 3, 4, 5, 6]>\n`

Join ([$] ,)

Convert the array to the string by the specified delimeter.
Usage: `array delimeter [$],`
Example: `1 2 3 3 ] "," [$], .NL`
Result: `1,2,3\n`

Outer product (#*word)

Does *word* on a1[y] and a2[x]
Usage: `a1 a2 #*word`
Example: `1 2 3 3 ] : #++ .NL`
Result: `<Array [<Array [2, 3, 4]>, <Array [3, 4, 5]>, <Array [4, 5, 6]>]>\n`

Reduce (#-*word*)

Calculates array[0] array[1] *word* ... array[n] *word*.
Usage: `array #-word`
Example: `1 2 3 3 ] #-+ .NL`
Result: `6\n`

Old map (#+*word*)

Extracts the array to the stack, and does *word* to each elements.
Usage: `array #+word`
Example: `wd-begin 1 + wd-end example 1 2 3 3 ] #+example .NL`
Result: `4\n`

Map (# : *word*)

Does *word* to each elements of the array.
Usage: `array #:word`
Example: `wd-begin 1 + wd-end example 1 2 3 3 ] #:example .NL`
Result: `<Array [2, 3, 4]>\n`

Chapter 6. Stack Operators

Duplication (:)

Duplicates the value.
Usage: `value :`
Example: `1 : . .`
Result: `11`

Pop (\)

Removes the top of the value of the stack.
Usage: `value \`
Example: `1 \ .NL`
Result: `StackUnderflow`

Pushes to the second stack (2<)

Pushes the value to the second stack.
Usage: `value 2<`
Example: `5 2< 2> .NL`
Result: `5\n`

Gets the value from the second stack (2>)

Duplicates the value.
Usage: `2>`
Example: `5 2< 2> .`
Result: `5`

Reverse (<>)

Reverses the values.
Usage: `values length <>`
Example: `1 2 2 <> .`
Result: `1`

Chapter 7. Expression Stack Operators

Push (>{})

Pushes the string as an expression to the expression stack.
Usage: `expr >{}`
Example: `"1 1 + .NL" >{} !{}`
Result: `2\n`

Pop (^{})

Removes the top of the expression of the expression stack.
Usage: `^{}`
Example: `"1 1 + .NL" >{} ^{} !{}`
Result: `StackUnderflow`

Run (!{})

Runs the top of the expression of the expression stack.
Usage: `!{}`
Example: `"1 1 + .NL" >{} !{}`
Result: `2\n`

Chapter 8. Comparison Operators

Equal to (`[EQ]`)

Pushes 1 if both values are equal.
Usage: `value1 value2 [EQ]`
Example: `1 1 [EQ] .NL`
Result: `1\n`

Not equal to (`[NE]`)

Pushes 1 if both values are not equal.
Usage: `value1 value2 [NE]`
Example: `1 0 [NE] .NL`
Result: `1\n`

Less than (`[LT]`)

Pushes 1 if value1 is less than value2.
Usage: `value1 value2 [LT]`
Example: `1 2 [LT] .NL`
Result: `1\n`

Less than or equal to (`[LE]`)

Pushes 1 if value1 is less than value2 or both are equal.
Usage: `value1 value2 [LE]`
Example: `1 1 [LE] .NL`
Result: `1\n`

Greater than (`[GT]`)

Pushes 1 if value1 is greater than value2.
Usage: `value1 value2 [GT]`
Example: `2 1 [GT] .NL`
Result: `1\n`

Greater than or equal to (`[GE]`)

Pushes 1 if value1 is greater than value2 or both are equal.
Usage: `value1 value2 [GE]`
Example: `1 1 [GE] .NL`
Result: `1\n`

Chapter 9. RegEx Operators

Generate (!/#/)

Generates a RegEx.
Usage: `pattern flag !/#/`
Example: `"0x[0-9a-fA-F]+" "g" !/#/ .NL`
Result: `<RegEx /0x[0-9a-fA-F]+/g>\n`

Match (?/#/)

Does pattern matching.
Usage: `string regex ?/#/`
Example: `"0xbf" "0x([0-9a-fA-F]+)" "" !/#/ ?/#/ 1 @ .NL`
Result: `bf\n`

Split (,/#/)

Splits the string by the RegEx.
Usage: `string regex ,/#/`
Example: `"1x2x3" "x" "g" !/#/ ,/#/ .NL`
Result: `<Array [1, 2, 3]>\n`

Replace (^/#/)

Replace the string by RegEx.
Usage: `string to regex ^/#/`
Example: `"0xbf" "[hex]" "0x[0-9a-fA-F]+" "g" !/#/ ^/#/ .NL`
Result: `[hex]\n`

Chapter 10. TCP Operators

Server (~#)

Pushes TCPServer.
Usage: `allowHalfOpen ~#`
Example: `0 ~# .NL`
Result: `<EventEmitter TCPServer>\n`

Client (~@)

Pushes TCPClient.
Usage: `allowHalfOpen ~@`
Example: `0 ~@ .NL`
Result: `<EventEmitter TCPSocket>\n`

Write (~>)

Writes the data to the socket.
Usage: `socket data ~>`
Example: `{ "Welcome!" ~> \ } 1 ~# "connection" >>!{} 8080 ~!`
Result: **See the picture in the page 20.**

End (~<)

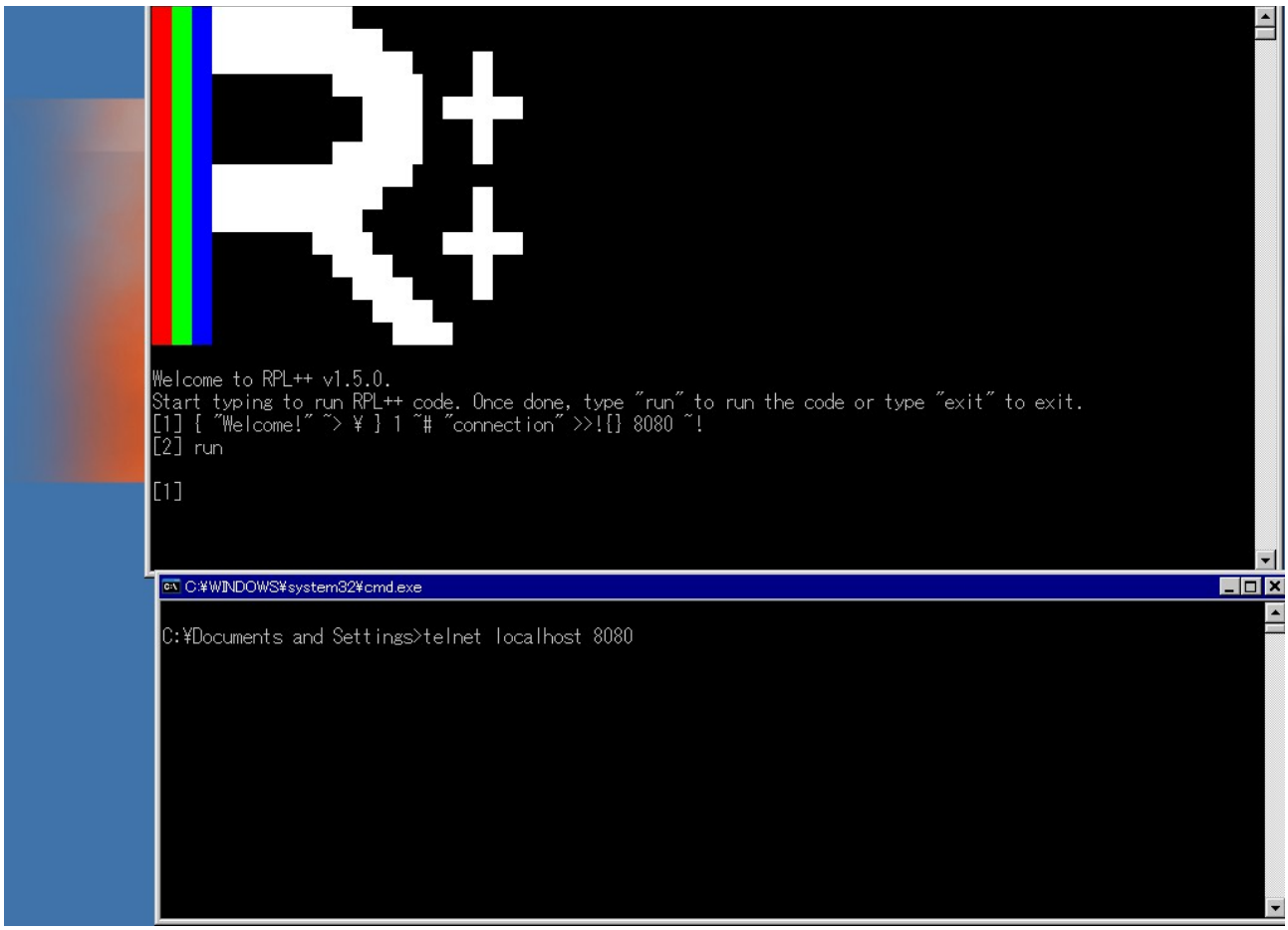
Writes the data to the socket, and sends FIN packet.
Usage: `socket data ~<`

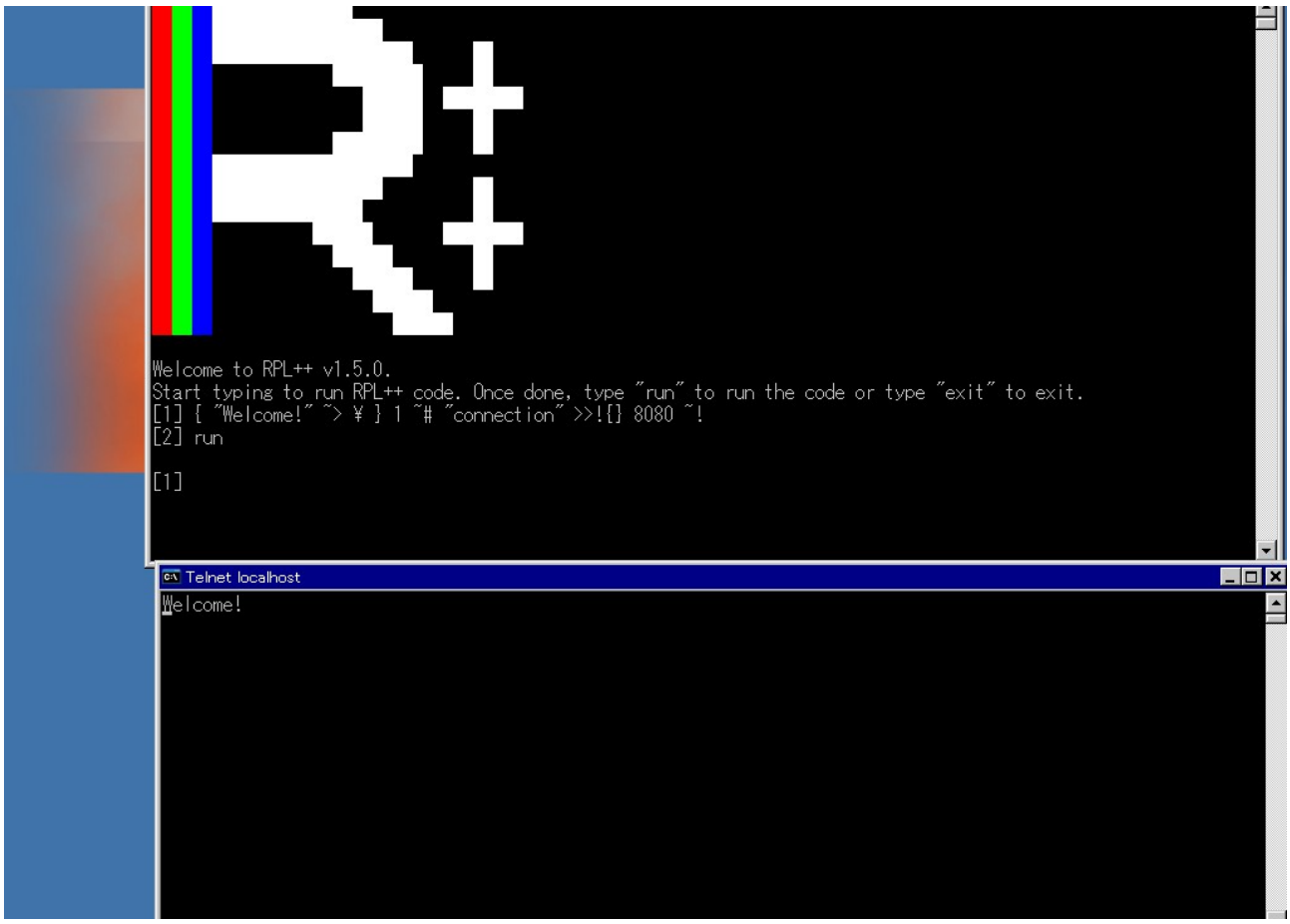
Listen (~!)

Listens on the port.
Usage: `server port ~!`
Example: `{ "Welcome!" ~> \ } 1 ~# "connection" >>!{} 8080 ~!`
Result: See the picture in the page 20.

Connect (~@!)

Connects to the server.
Usage: `socket host port ~@!`


Welcome to RPL++ v1.5.0.
Start typing to run RPL++ code. Once done, type "run" to run the code or type "exit" to exit.
[1] ["Welcome!" ~> ¥] 1 ~# "connection" >>![] 8080 ~!
[2] run
[1]
C:¥WINDOWS¥system32¥cmd.exe
C:¥Documents and Settings>telnet localhost 8080


Welcome to RPL++ v1.5.0.
Start typing to run RPL++ code. Once done, type "run" to run the code or type "exit" to exit.
[1] ["Welcome!" ~> ¥] 1 ~# "connection" >>![] 8080 ~!
[2] run
[1]
Telnet localhost
Welcome!

Chapter 11. Event Operators

Emit (>?*name*)

Emits the event.
Usage: `eventListener >?name`
Example: `{ "Hello!" .NL } >!test "hello" >>!{} >?hello`
Result: `Hello!\n`

Generates an event listener (>!*name*)

Generates an event listener.
Usage: `>!name`
Example: `{ "Hello!" .NL } >!test "hello" >>!{} >?hello`
Result: `Hello!\n`

Listen on (>>*listener*)

Listen on the event.
Note(1): TCPSocket will be pushed to the stack when connection event is called in TCPServer.
Note(2): String will be pushed to the stack when data event is called in TCPServer / TCPClient.
Note(3): Variable #SELF will contain itself (event listener).
Usage: `eventListener name >>listener`
Example: `{ "Hello!" .NL } >!test "hello" >>!{} >?hello`
Result: `Hello!\n`

Chapter 12. Misc Operators

Define a variable (=)

Define a variable.
Note: If you define a variable with name that starts with % and ends with %, it'll be defined as a variable that can be used in the local word (`wd-end_`).
Usage: `data name =`
Example: `"Hello!" "hi" = hi .NL`
Result: `Hello!\n`

String to a number ($)

Converts string to a number.
Usage: `string $`
Example: `"0xbf" $ .NL`
Result: `191\n`

Number to a character (U+xxxx) ($>?)

Converts number as code point to a character.
Usage: `code $>?`
Example: `0x21 $>? .NL`
Result: `!\n`

Comeback (:<)

Usage: `:<`
Example:
```
;hello "Hello " . :<
:>hello "world!" .NL
```
Result: `Hello world!\n`

Closes the switch statement (=.)

Closes the switch statement.
Usage: `=.`
Example:
```
1 1 =(
  "one" .NL
). 2 =>(
  "two" .NL
). =.
```
Result: `one\n`

Do nothing (--)

Does nothing.
Usage: `--`
Example: `--`
Result: (nothing happens)

Converts to a Buffer (>>>)

Converts to a Buffer.
Usage: `array >>>`
Usage: `number >>>`
Example: `0x21 >>> .NL`
Result: `<Buffer [21]>\n`

Runs JavaScript (.JS)

Runs JavaScript.
Usage: `program .JS`
Example: `"stack.push(1)" .JS .NL`
Result: `1\n`

Goto (#)

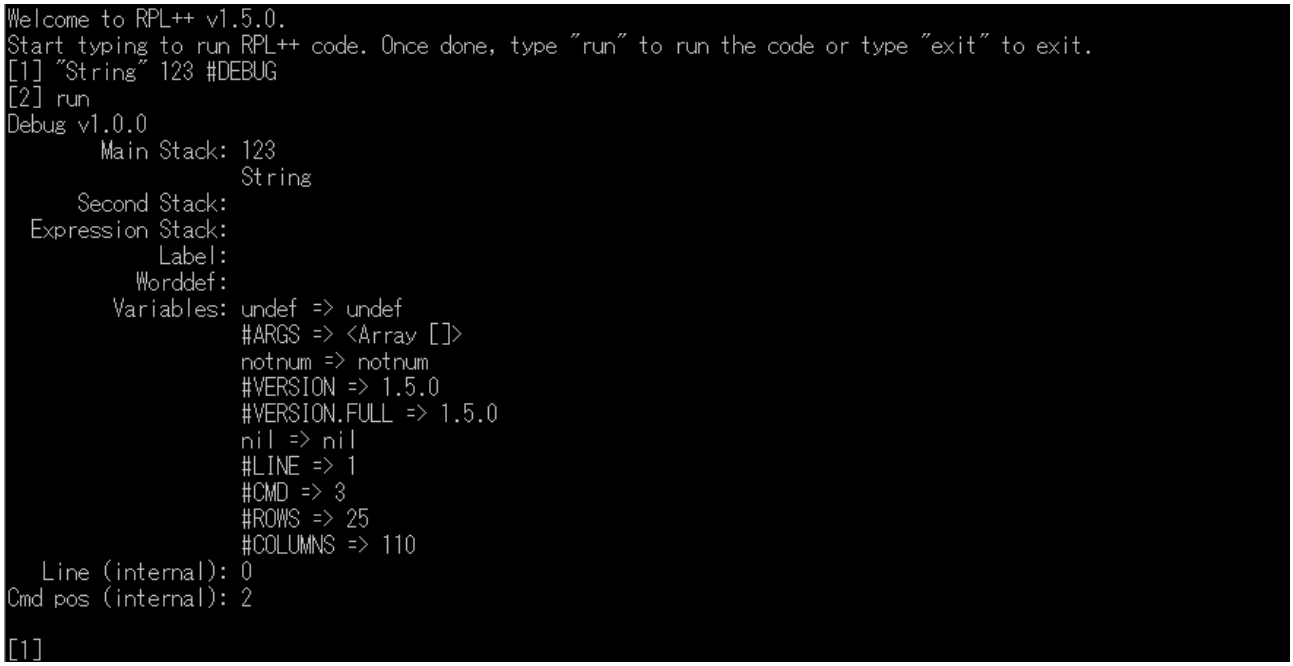
Go to line:command.
Usage: `line command #`
Example:
```
3 1 #
"This line won't get ran" .NL
"End" .NL
```
Result: `End\n`

Goto with the condition (;)

Go to line:command with the condition.
Usage: `condition line command ;`
Example:
```
1 3 1 ;
"This line won't get ran" .NL
"End" .NL
```
Result: `End\n`

Debug (#DEBUG)

Usage: `#DEBUG`
Example: `"String" 123 #DEBUG`
Result:



Chapter 13. Library

Standard Libraries

MathEx – Mathemetical things
wavefile.write – Wavefile Writer
http – HTTP Server

Import (`.IM`)

Imports the library.
Usage: `filename .IM`

Imports the standard library (`.IMS`)

Imports the standard library.
Usage: `libraryName .IMS`

Chapter 14. Label

Define (`:name`)

Defines the label.
Usage: `:name`
Example:
```
0 "I" =
        :label I 1 + : "I" = .NL
        "Reached here" .NL >label
```
Result: `0\nReached here\n1\n2\n3\n4\n5\n...`

Define (`;name`)

Defines the label.
Usage: `;name`
Example:
```
0 "I" =
        ;label I 1 + : "I" = .NL
        "Reached here" .NL >label
```
Result: `Reached here\n0\n1\n2\n3\n4\n5\n...`

Jump (`>name`)

Jumps to the label.
Usage: `>name`
Example:
```
0 "I" =
        ;label I 1 + : "I" = .NL
        "Reached here" .NL >label
```
Result: `Reached here\n0\n1\n2\n3\n4\n5\n...`

Jump with the condition (`?>name`)

Jumps to the label with the condition.
Usage: `?>name`
Example:
```
0 "I" =
        ;label I 1 + : "I" = .NL
        "Reached here" .NL 0 ?>label
```
Result: `Reached here\n`

Jump, but with the recursive mode (:>*name*)

Jumps to the label, but with the recursive mode.
Usage: `:>name`
Example:
```
0 "I" =
;label "Hello!" .NL :<
:>label
```
Result: `Hello!\n...`

Jump, with the condition but with the recursive mode (?:>*name*)

Jumps to the label, with the condition but with the recursive mode.
Usage: `?:>name`
Example:
```
0 "I" =
;label "Hello!" .NL :<
0 ?:>label
```
Result: (no output)

Post-label (_*name*_)

Defines a post-label.
Usage: `_name_`
Example:
```
_>label
"Hello!" .NL :<
_label_
```
Result: (no output)

Jumps to the post-label (_>*name*)

Jumps to the post-label.
Usage: `_>name`
Example:
```
_>label
"Hello!" .NL :<
_label_
```
Result: (no output)

Jumps to the post-label with the condition (_?>*name*)

Jumps to the post-label with the condition.
Usage: `condition _?>name`
Example:
```
1 _?>label
"Hello!" .NL :<
_label_
```
Result: (no output)

Begins worddef (wd-begin)

Begins worddef.
Usage: wd-begin
Example:
```
wd-begin
  1 +
wd-end(1) word
1 word .NL
```
Result: 2\n

Ends worddef (wd-end)

Ends worddef.
Usage:
```
wd-end name
wd-end(min_args) name
wd-end_ name
wd-end_(func1,func2...funcn) name
wd-end_(func1,func2...funcn)(min_args) name
```
Example1:
```
wd-begin
  1 +
wd-end(1) word
1 word .NL
```
Result1: 2\n
Example2:
```
wd-begin
  a
wd-end_ word
0 "a" = word
```
Result2: UnknownWord
Example3:
```
wd-begin
  "Hello!"
wd-end a
wd-begin
  a .NL
wd-end_ word
word
```
Result3: UnknownWord

Example4:
```
wd-begin
  "Hello!"
wd-end a
wd-begin
  a .NL
wd-end_(a) word
word
```
Result4: `Hello!\n`

If statement (`?( ).`)

Runs the code if the condition is 1.
Usage: `condition ?( code ).`
Example:
```
1 ?(
  "Hello!" .NL
).
```
Result: `Hello!\n`

If not statement (`?!( ).`)

Runs the code if the condition is not 1.
Usage: `condition ?!( code ).`
Example:
```
0 ?!(
  "Hello!" .NL
).
```
Result: `Hello!\n`

Switch statement

Note: First “case” of switch statements must be =(

Usage:
```
v1 v2 =( code ). v3 =>( code ). ... =.
```

Example:
```
0 "x" =
x 0 =(
  "x is 0" .NL
). 1 =>(
  "x is 1" .NL
). =.
```

Result:
```
x is 0\n
```

While statement (* ( ) .)

Usage:
```
{ expr } *( code ).
```

Example:
```
0 "I" =
{ I 10 [LT] } *(
  I .NL
  I 1 + "I" =
).
```

Result:
```
0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n
```

Try-catch statement

Usage:
```
%( try ):( catch )%
%( try ):[error]( catch )%
```

Example:
```
%(
  a
):(
  .NL
)%
```

Result:
```
<Array [UnknownWord, UnknownWord]>\n
```

Note: First element means “internal name”, and second one means “user-defined name”

Chapter 16. Environment Variables

undef

Undefined.

notnum

Not a number.
[NE] pushes 1 when notnum and notnum are compared.

nil

nil.

#ARGS

Contains arguments that is passed to the interpreter.

#LINE

Contains the line number that is currently ran.

#CMD

Contains the row number that is currently ran.

#ROWS

Contains the row length of the console.

#COLUMNS

Contains the column length of the console.

#VERSION

Contains the version of RPL++ .

#VERSION.FULL

Contains the full version of RPL++.

InternalError

(Used inside)

StackUnderflow

It occurs when an operand requires more values on the stack than are currently there.

UnknownWord

It is thrown when the RPL++ interpreter finds a token and cannot figure out what it means.

IncorrectType

It is thrown when an operand requires a specific type and you pass it the wrong type.

RPL++ Team

Nishi

TerraMaster85

c0repwn3r

GhostlyCoding