# Node.js v10.9.0 Documentation

Index | View on single page | View as JSON | View another version ▼ | Edit on GitHub

## Table of Contents

# Readline #

Stability: 2 - Stable

The `readline` module provides an interface for reading data from a Readable stream (such as `process.stdin`) one line at a time. It can be accessed using:

```
const readline = require('readline');
```

The following simple example illustrates the basic use of the `readline` module.

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('What do you think of Node.js? ', (answer) => {
  // TODO: Log the answer in a database
  console.log(`Thank you for your valuable feedback: ${answer}`);

  rl.close();
});
```

Once this code is invoked, the Node.js application will not terminate until the `readline.Interface` is closed because the interface waits for data to be received on the `input` stream.

## Class: Interface #

Added in: v0.1.104

Instances of the `readline.Interface` class are constructed using the `readline.createInterface()` method. Every instance is associated with a single `input` Readable stream and a single `output` Writable stream. The `output` stream is used to print prompts for user input that arrives on, and is read from, the `input` stream.

## Event: 'close' #

Added in: v0.1.98

The `'close'` event is emitted when one of the following occur:

- The `rl.close()` method is called and the `readline.Interface` instance has relinquished control over the `input` and `output` streams;
- The `input` stream receives its `'end'` event;
- The `input` stream receives `<ctrl>-D` to signal end-of-transmission (EOT);
- The `input` stream receives `<ctrl>-C` to signal `SIGINT` and there is no `'SIGINT'` event listener registered on the `readline.Interface` instance.

The listener function is called without passing any arguments.

The `readline.Interface` instance is finished once the `'close'` event is emitted.

## Event: 'line' #

Added in: v0.1.98

The `'line'` event is emitted whenever the `input` stream receives an end-of-line input (`\n`, `\r`, or `\r\n`). This usually occurs when the user presses the `<Enter>`, or `<Return>` keys.

The listener function is called with a string containing the single line of received input.

```
rl.on('line', (input) => {
  console.log(`Received: ${input}`);
});
```

## Event: 'pause' #

Added in: v0.7.5

The `'pause'` event is emitted when one of the following occur:

- The `input` stream is paused.
- The `input` stream is not paused and receives the `'SIGCONT'` event. (See events `'SIGTSTP'` and `'SIGCONT'`.)

The listener function is called without passing any arguments.

```
rl.on('pause', () => {
  console.log('Readline paused.');
});
```

## Event: 'resume' #

Added in: v0.7.5

The `'resume'` event is emitted whenever the `input` stream is resumed.

The listener function is called without passing any arguments.

```
rl.on('resume', () => {
  console.log('Readline resumed.');
});
```

## Event: 'SIGCONT' #

Added in: v0.7.5

The `'SIGCONT'` event is emitted when a Node.js process previously moved into the background using `<ctrl>-Z` (i.e. `SIGTSTP`) is then brought back to the foreground using `fg(1p)`.

If the `input` stream was paused *before* the `SIGTSTP` request, this event will not be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGCONT', () => {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```

The `'SIGCONT'` event is *not* supported on Windows.

## Event: 'SIGINT' #

Added in: v0.3.0

The `'SIGINT'` event is emitted whenever the `input` stream receives a `<ctrl>-C` input, known typically as `SIGINT`. If there are no `'SIGINT'` event listeners registered when the `input` stream receives a `SIGINT`, the `'pause'` event will be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGINT', () => {
  rl.question('Are you sure you want to exit? ', (answer) => {
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```

## Event: 'SIGTSTP' #

Added in: v0.7.5

The `'SIGTSTP'` event is emitted when the `input` stream receives a `<ctrl>-Z` input, typically known as `SIGTSTP`. If there are no `'SIGTSTP'` event listeners registered when the `input` stream receives a `SIGTSTP`, the Node.js process will be sent to the background.

When the program is resumed using `fg(1p)`, the `'pause'` and `'SIGCONT'` events will be emitted. These can be used to resume the `input` stream.

The `'pause'` and `'SIGCONT'` events will not be emitted if the `input` was paused before the process was sent to the background.

The listener function is invoked without passing any arguments.

```
rl.on('SIGTSTP', () => {
  // This will override SIGTSTP and prevent the program from going to the
  // background.
  console.log('Caught SIGTSTP.');
});
```

The `'SIGTSTP'` event is *not* supported on Windows.

## rl.close() #

Added in: v0.1.98

The `rl.close()` method closes the `readline.Interface` instance and relinquishes control over the `input` and `output` streams. When called, the `'close'` event will be emitted.

## rl.pause() #

Added in: v0.3.4

The `rl.pause()` method pauses the `input` stream, allowing it to be resumed later if necessary.

Calling `rl.pause()` does not immediately pause other events (including `'line'`) from being emitted by the `readline.Interface` instance.

## rl.prompt([preserveCursor]) #

Added in: v0.1.98

- `preserveCursor` <boolean> If `true`, prevents the cursor placement from being reset to `0`.

The `rl.prompt()` method writes the `readline.Interface` instances configured `prompt` to a new line in `output` in order to provide a user with a new location at which to provide input.

When called, `rl.prompt()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the prompt is not written.

## rl.question(query, callback) #

Added in: v0.3.3

- `query` <string> A statement or query to write to `output`, prepended to the prompt.
- `callback` <Function> A callback function that is invoked with the user's input in response to the `query`.

The `rl.question()` method displays the `query` by writing it to the `output`, waits for user input to be provided on `input`, then invokes the `callback` function passing the provided input as the first argument.

When called, `rl.question()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the `query` is not written.

Example usage:

```
rl.question('What is your favorite food? ', (answer) => {
  console.log(`Oh, so your favorite food is ${answer}`);
});
```

The `callback` function passed to `rl.question()` does not follow the typical pattern of accepting an `Error` object or `null` as the first argument. The `callback` is called with the provided answer as the only argument.

## rl.resume() #

Added in: v0.3.4

The `rl.resume()` method resumes the `input` stream if it has been paused.

## rl.setPrompt(prompt) #

Added in: v0.1.98

- `prompt` `<string>`

The `rl.setPrompt()` method sets the prompt that will be written to `output` whenever `rl.prompt()` is called.

## rl.write(data[, key]) #

Added in: v0.1.98

- `data` `<string>`
- `key` `<Object>`
    - `ctrl` `<boolean>` `true` to indicate the `<ctrl>` key.
    - `meta` `<boolean>` `true` to indicate the `<Meta>` key.
    - `shift` `<boolean>` `true` to indicate the `<Shift>` key.
    - `name` `<string>` The name of the a key.

The `rl.write()` method will write either `data` or a key sequence identified by `key` to the `output`. The `key` argument is supported only if `output` is a TTY text terminal.

If `key` is specified, `data` is ignored.

When called, `rl.write()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the `data` and `key` are not written.

```
rl.write('Delete this!');
// Simulate Ctrl+u to delete the line written previously
rl.write(null, { ctrl: true, name: 'u' });
```

The `rl.write()` method will write the data to the `readline` `Interface`'s `input` *as if it were provided by the user*.

## readline.clearLine(stream, dir) #

Added in: v0.7.7

- `stream` <stream.Writable>
- `dir` <number>
  - `-1` - to the left from cursor
  - `1` - to the right from cursor
  - `0` - the entire line

The `readline.clearLine()` method clears current line of given TTY stream in a specified direction identified by `dir`.

## readline.clearScreenDown(stream) #

Added in: v0.7.7

- `stream` <stream.Writable>

The `readline.clearScreenDown()` method clears the given TTY stream from the current position of the cursor down.

## readline.createInterface(options) #

▶ History

- `options` <Object>
  - `input` <stream.Readable> The Readable stream to listen to. This option is *required*.
  - `output` <stream.Writable> The Writable stream to write readline data to.
  - `completer` <Function> An optional function used for Tab autocompletion.
  - `terminal` <boolean> `true` if the `input` and `output` streams should be treated like a TTY, and have ANSI/VT100 escape codes written to it. **Default:** checking `isTTY` on the `output` stream upon instantiation.
  - `historySize` <number> Maximum number of history lines retained. To disable the history set this value to `0`. This option makes sense only if `terminal` is set to `true` by the user or by an internal `output` check, otherwise the history caching mechanism is not initialized at all. **Default:** `30`.

   - `prompt` <string> The prompt string to use. **Default:** `'> '`.
   - `crlfDelay` <number> If the delay between `\r` and `\n` exceeds `crlfDelay` milliseconds, both `\r` and `\n` will be treated as separate end-of-line input. `crlfDelay` will be coerced to a number no less than `100`. It can be set to `Infinity`, in which case `\r` followed by `\n` will always be considered a single newline (which may be reasonable for reading files with `\r\n` line delimiter). **Default:** `100`.
   - `removeHistoryDuplicates` <boolean> If `true`, when a new input line added to the history list duplicates an older one, this removes the older line from the list. **Default:** `false`.

The `readline.createInterface()` method creates a new `readline.Interface` instance.

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

Once the `readline.Interface` instance is created, the most common case is to listen for the `'line'` event:

```
rl.on('line', (line) => {
  console.log(`Received: ${line}`);
});
```

If `terminal` is `true` for this instance then the `output` stream will get the best compatibility if it defines an `output.columns` property and emits a `'resize'` event on the `output` if or when the columns ever change (`process.stdout` does this automatically when it is a TTY).

## Use of the `completer` Function #

The `completer` function takes the current line entered by the user as an argument, and returns an `Array` with 2 entries:

- An `Array` with matching entries for the completion.
- The substring that was used for the matching.

For instance: `[[substr1, substr2, ...], originalsubstring]`.

```
function completer(line) {
  const completions = '.help .error .exit .quit .q'.split(' ');
```

```
    const hits = completions.filter((c) => c.startsWith(line));
    // show all completions if none found
    return [hits.length ? hits : completions, line];
  }
```

The `completer` function can be called asynchronously if it accepts two arguments:

```
  function completer(linePartial, callback) {
    callback(null, [['123'], linePartial]);
  }
```

## readline.cursorTo(stream, x, y) #

Added in: v0.7.7

- `stream` <stream.Writable>
- `x` <number>
- `y` <number>

The `readline.cursorTo()` method moves cursor to the specified position in a given TTY `stream`.

## readline.emitKeypressEvents(stream[, interface]) #

Added in: v0.7.7

- `stream` <stream.Readable>
- `interface` <readline.Interface>

The `readline.emitKeypressEvents()` method causes the given Readable stream to begin emitting `'keypress'` events corresponding to received input.

Optionally, `interface` specifies a `readline.Interface` instance for which autocompletion is disabled when copy-pasted input is detected.

If the `stream` is a TTY, then it must be in raw mode.

This is automatically called by any readline instance on its `input` if the `input` is a terminal. Closing the `readline` instance does not stop the `input` from emitting `'keypress'` events.

```
readline.emitKeypressEvents(process.stdin);
if (process.stdin.isTTY)
  process.stdin.setRawMode(true);
```

## readline.moveCursor(stream, dx, dy) #

Added in: v0.7.7

- `stream` <stream.Writable>
- `dx` <number>
- `dy` <number>

The `readline.moveCursor()` method moves the cursor *relative* to its current position in a given TTY `stream`.

## Example: Tiny CLI #

The following example illustrates the use of `readline.Interface` class to implement a small command-line interface:

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
  prompt: 'OHAI> '
});

rl.prompt();

rl.on('line', (line) => {
  switch (line.trim()) {
    case 'hello':
      console.log('world!');
      break;
    default:
      console.log(`Say what? I might have heard '${line.trim()}'`);
      break;
  }
  rl.prompt();
}).on('close', () => {
  console.log('Have a great day!');
  process.exit(0);
});
```

## Example: Read File Stream Line-by-Line #

A common use case for `readline` is to consume input from a filesystem Readable stream one line at a time, as illustrated in the following example:

```
const readline = require('readline');
const fs = require('fs');

const rl = readline.createInterface({
  input: fs.createReadStream('sample.txt'),
  crlfDelay: Infinity
});

rl.on('line', (line) => {
  console.log(`Line from file: ${line}`);
});
```