Apr 18, 2021

CRLF vs. LF: Normalizing Line Endings in Git

#git, #operating-systems, #tooling

If you've ever worked on a project where developers use different operating systems, you know that line endings can be a peculiar source of frustration. This issue of CRLF vs. LF line endings is actually fairly popular—you'll find tons of questions on StackOverflow about how to configure software like git to play nicely with different operating systems.

The typical advice is to configure your local git to handle line ending conversions for you. We'll look at how that can be done in this article, but it isn't ideal if you're on a large team of developers. For example, if just one person forgets to configure their line endings correctly, you'll need to re-normalize your line endings and recommit your files every time a change is made.

A better solution is to add a `.gitattributes` file to your repo so you can enforce line endings consistently in your codebase regardless of what operating systems your developers are using. Before we look at how this is done, we'll briefly review the history of line endings on Windows vs. Unix so we can understand why this issue exists in the first place.

History can be boring, though, so if you stumbled upon this post after hours of frustrated research, you can skip straight to A Simple `.gitattributes` Config and grab the code. However, I do encourage reading the full post to understand how these things work under the hood—you'll (hopefully) never have to Google line endings again!

Table of Contents

CRLF vs. LF: What Are Line Endings, Anyway?

To *really* understand the problem of CRLF vs. LF line endings, we need to brush up on a bit of typesetting history.

People use letters, numbers, and symbols to communicate with one another. It's how you're reading this post right now! But computers can only understand and work with *numbers*. Since the files on your computer consist of strings of human-readable characters, we need a system that allows us to convert back and forth between these two formats. The ASCII standard is that system—it maps characters like `A` and `z` to numbers, bridging the gap between human languages and the language of computers.

Interestingly, the ASCII standard isn't just for *visible* characters like letters and numbers. A certain subset are **control characters**, also known as **non-printing characters**. They aren't used to render visible characters; rather, they're used to perform unique actions, like deleting the previous character or inserting a newline.

`LF` and `CR` are two such control characters, and they're both related to line endings in files. Their history dates back to the era of the typewriter, so we'll briefly look at how that works so you understand why we have two different control characters rather than just one. Then, we'll look at how this affects the typical developer experience on a multi-OS codebase.

`LF`: Line Feed

LF stands for "line feed," but you're probably more familiar with the term **newline** (the escape sequence `\n`). Simply put, this character represents the end of a line of text. On Linux and Mac, this is equivalent to the start of a new line of text. That distinction is important because Windows does not follow this convention. We'll

`CR`: Carriage Return

CR (the escape sequence `\r`) stands for **carriage return**, which moves the cursor to the start of the current line. If you've ever seen a download progress bar on your terminal, this is how it works its magic! By using the carriage return, your terminal can animate text in place by returning the cursor to the start of the current line and overwriting any previously rendered text.

That's cool, but you may be wondering where the need for such a character originated (beyond just animating text, which happens to be a niche application). It's a good question—and the answer will help us better understand why Windows uses `CRLF`.

Typewriters and the Carriage Return

Back when dinosaurs roamed the earth, people used to lug around these chunky devices called *typewriters*.

Photo credit: Patrick Fore, Unsplash

You feed the device a sheet of paper fastened to a mechanical roll known as the **carriage**. With each keystroke, the typewriter prints letters using ink on your sheet of paper, shifting the carriage to the left to ensure that the next letter you type will appear to the right of the previous one. You can watch a typewriter being used in action to get a better sense for how this works.

Of course, once you run out of space on the current line, you'll need to go down to the next line on your sheet of paper. This is done by rotating the carriage to move the paper up a certain distance relative to the typewriter's "pen." But you also need to reset your carriage so that the next character you type will be aligned to the left-hand margin of your paper. In other words, you need some way to *return* the carriage to its starting position. And that's precisely the job of the **carriage return**: a metal lever attached to the left side of the carriage that, when pushed, returns the carriage to its starting position.

> **Fun Fact**: You may be familiar with the characteristic *ding* sound that a typewriter makes from movies or video games (thanks, *Dishonored*!). This is known as the margin bell, which is triggered as soon as your text approaches the very right margin to signal that you need to move on to the next line.

That's all good and well, but you're probably wondering how this is relevant in the world of computers, where carriages, levers, and all these contraptions seem obsolete. We're getting there!

Teletypewriters and the Birth of `CRLF`

Moving on to the early 20th century, we arrive at the **teletypewriter**, yet another device predating the modern computer. Basically, it works exactly the same way that a message to a receiving party via a transmitter, either over a physical wire or radio waves.

Now we're digital! These devices needed to use both a line feed character (`LF`) and a carriage return character (`CR`) to allow you to type from the start of the next line of text. That's exactly how the original typewriter worked, except it didn't have any notion of "characters" because it was a mechanically operated device. With the teletype, this process is more or less automatic and triggered by a keystroke—you don't have to manually push some sort of "carriage" or move a sheet of paper up or down to achieve the same effect.

It's easier to visualize this if you think of `LF` and `CR` as representing independent movements in either the horizontal or vertical direction, but not both. By itself, a line feed moves you down vertically; a carriage return resets your "cursor" to the very start of the current line. We saw the physical analogue of `CR` and `LF` with typewriters—moving to the next line of text required rotating the carriage to move the sheet of paper up (line feed), and returning your "cursor" to the start of that new line required using a mechanical piece aptly named the *carriage return*.

Teletypes set the standard for `CRLF` line endings in some of the earliest operating systems, like the popular MS-DOS. Microsoft has an excellent article explaining the history of `CRLF` in teletypes and early operating systems. Here's a relevant snippet:

> *“This protocol dates back to the days of teletypewriters. CR stands for "carriage return" – the CR control character returned the print head ("carriage") to column 0 without advancing the paper. LF stands for "linefeed" – the LF control character advanced the paper one line without moving the print head. So if you wanted to return the print head to column zero (ready to print the next line) and advance the paper (so it prints on fresh paper), you need both CR and LF.*
>
> *If you go to the various internet protocol documents, such as RFC 0821*

> *question is not "Why do CP/M, MS-DOS, and Win32 use CR+LF as the line terminator?" but rather "Why did other people choose to differ from these standards documents and use some other line terminator?"”*
>
> —*Why is the line terminator CR+LF?*

MS-DOS used the two-character combination of `CRLF` to denote line endings in files, and modern Windows computers continue to use `CRLF` as their line ending to this day. Meanwhile, from its very inception, Unix used `LF` to denote line endings, ditching `CRLF` for consistency and simplicity. Apple originally used only `CR` for Mac Classic but eventually switched to `LF` for OS X, consistent with Unix.

This makes it seem like Windows is the odd one out when it's *technically* not. Developers usually get frustrated with line endings on Windows because `CRLF` is seen as an artifact of older times, when you actually *needed* both a carriage return and a line feed to represent newlines on devices like teletypes.

It's easy to see why `CRLF` is redundant by today's standards—using both a carriage return and a line feed assumes that you're bound to the physical limitations of a typewriter, where you *had* to explicitly move your sheet of paper up and then reset the carriage to the left-hand margin. With a file, it suffices to define the newline character as implicitly doing the job of both a line feed and a carriage return under the hood. In other words, so long as your operating system defines the newline character to mean that the next line starts at the *beginning* and not at some arbitrary column offset, then we have no need for an explicit carriage return *in addition* to a line feed—one symbol can do the job of both.

While it may seem like a harmless difference between operating systems, this issue of CRLF vs. LF has been causing people headaches for a long time now. For example, basic Windows text editors like Notepad used to not be able to properly interpret `LF` alone as a true line ending. Thus, if you opened a file created on Linux or Mac with Notepad, the line endings would not get rendered correctly. Notepad was later updated

Inspecting and Converting Line Endings (in Bash)

Cool! Now that we know how line endings originated, it's worth learning something practical.

In bash (or, equivalently, WSL if you're using Windows), you can view line endings for a specific file using `cat` with the `A` flag:

```
cat -A myFile
```

If a file is using `CRLF`, you'll see the string `^M$` at the end of each line, where `^M` denotes a carriage return and `$` a line feed. Here's an example of what that might look like:

```
line one^M$
line two^M$
line three^M$
```

If a file is using `LF`, then you'll only see dollar signs:

```
line one$
line two$
line three$
```

You can also use the `dos2unix` command-line utility to convert a file from DOS format (Windows, using `CRLF`) to UNIX (`LF`).

Line Endings in Git

Admittedly, that was a lot of background to get through! But it was worth it because we're finally ready to talk about line endings as they relate to git (and how to solve the problem of `CRLF` vs. `LF` in any given code base).

As you can probably guess, the lack of a universal line ending presents a dilemma for software like git, which relies on very precise character comparisons to determine if a file has changed since the last time it was checked in. If one developer uses Windows and another uses Mac or Linux, and they each save and commit the same files, they may see line ending changes in their git diffs—a conversion from `CRLF` to `LF` or vice versa. This leads to unnecessary noise due to single-character changes and can be quite annoying.

Configuring Line Endings in Git with `core.autocrlf`

You can tell git how you'd like it to handle line endings on your system with the `core.autocrlf` configuration. This is done with the following command:

```
git config --global core.autocrlf [true|false|input]
```

Note that a value of `false` turns off any line ending conversions, which is usually undesirable unless you know that everyone on your team is using the same OS. That's rarely the case, so you can simply forget that this option even exists (unless you're using

That leaves us with just two options: `autocrlf true` and `autocrlf input`. What's the difference between these two?

`autocrlf true`

With `autocrlf true`, files will be checked out as `CRLF` locally with git, but whenever you commit files, all instances of `CRLF` will be replaced with `LF`. Basically, this setting ensures that your codebase always uses `LF` in the final version of all files but `CRLF` locally when checked out. This is the recommended setting for Windows developers since `CRLF` is the native line ending for Windows.

If you use this option, you may see this warning when staging files for a commit on Windows:

```
warning: CRLF will be replaced by LF in <file-name>.
The file will have its original line endings in your working directory.
```

This doesn't mean that something went wrong, so there's no need to panic. Git is just warning you that your `CRLF` line endings will be normalized to `LF` on commit, per this setting's intended behavior.

`autocrlf input`

With `autocrlf input`, files are converted to `LF` when they get committed, but they are not converted to anything when checked out. Hence the name "input"—you get what you originally put in. If a file was originally committed as `CRLF` on accident by a Windows developer, you'll see it as `CRLF` locally (and if you modify it, you'll force it back to `LF`). If a file was originally added as `LF`, you'll see it as such. This is usually a

The only difference between this option and `true` is that `input` doesn't touch line endings on checkout. This is the recommended setting for Mac/Linux developers since those operating systems use `LF` by default.

Normalizing Line Endings in Git with `.gitattributes`

You certainly *can* ask all your developers to configure their local git according to a chosen standard (`autocrlf true` or `autocrlf input`). But this is tedious and highly error prone, and it can be confusing trying to recall what these options mean since their recommended usage depends on what operating system you're on. If a developer installs a new environment or gets a new laptop, they'll need to remember to reconfigure git. And if a new developer forgets to read your docs, or a one-off developer from another team contributes to your repo, then you'll start seeing line ending changes again. Not good.

Fortunately, there's a better solution: creating a `.gitattributes` file at the root of your repo to settle things once and for all. Git reads this file and applies its rules whenever you check out or commit files, ensuring that your line ending conventions are enforced regardless of how each individual developer has configured git locally or what OS they're using.

A Simple `.gitattributes` Config

Here's a `.gitattributes` file that should cover most use cases:

Filename: .gitattributes

```
* text=auto eol=lf

# Isolate binary files in case the auto-detection algorithm fails and
# marks them as text files (which could brick them).
*.{png,jpg,jpeg,gif,webp,woff,woff2} binary
```

Commit the file and push it to your remote.

You can learn more about how this works in the answer to this StackOverflow question: What's the difference between “* text=auto eol=lf” and “* text eol=lf” in .gitattributes?. I've provided a condensed summary below.

First, you need to understand that git uses a simple algorithm to detect whether a particular file in your repo is a text file or a binary file (e.g., an executable, image, or font file). By default, this algorithm is used for the purpose of diffing files that have changed, but it can also come in handy for the purpose of enforcing line ending conventions (as we're doing here).

That's what `text=auto` does in the config above—it tells git to apply its auto-detection algorithm to determine whether a file is a text file. Then, `eol=lf` tells git to enforce `LF` line endings for text files **on both checkout and commit**. This should work well on both Windows and Linux since a majority of cross-platform text editors support `LF`. (But as I mentioned earlier, historically, some Windows text editors like Notepad would struggle to interpret `LF` alone. Nowadays, this is less of a problem.)

Git's auto-detection algorithm is fairly accurate, but in case it fails to correctly distinguish between a text file and a binary file (like an image or font file), we can also explicitly mark a subset of our files as binary files to avoid bricking them. That's what we're doing here:

Now, after committing this file, the final step is to renormalize all your line endings for any files that were checked into git **prior** to the addition of `.gitattributes`. You can do that with the following command since git 2.16:

```
git add --renormalize .
```

This reformats all your files according to the rules defined in your `.gitattributes` config. If previously committed files are using `CRLF` in git's index and are converted to `LF` as a result of this renormalization, their line endings will be updated in the index, and those files will be staged for a commit. The only thing left to do is to commit those changes and push them to your repo. From that point onward, anytime a new file is introduced, its line endings will be checked in (and checked out) as `LF`.

Git Line Endings: Working Tree vs. Index

You may see the following message when you commit these renormalized files:

```
warning: CRLF will be replaced by LF in <file-name>.
The file will have its original line endings in your working directory.
```

This is the expected behavior— `CRLF` will become `LF` in Git's index, meaning when you push those files to your repo, they'll have `LF` line endings in the remote copy of your code. Anyone who later pulls or checks out that code will see `LF` line endings locally.

But git doesn't actually change line endings for the **local copies** of your files (i.e., the

Rest assured that these files will never use `CRLF` in the *remote* copy of your code if you've specified `LF` as your desired line ending convention in the `.gitattributes` file.

Verifying Line Endings in Git for Any File

If you want to double-check that the files in Git's index are using the correct line endings after all of these steps, you can run the following command:

```
git ls-files --eol
```

This will show you line ending information for all files that git is tracking, in a format like this:

```
i/lf    w/crlf  attr/text=auto eol=lf   file.txt
```

From left to right, these are:

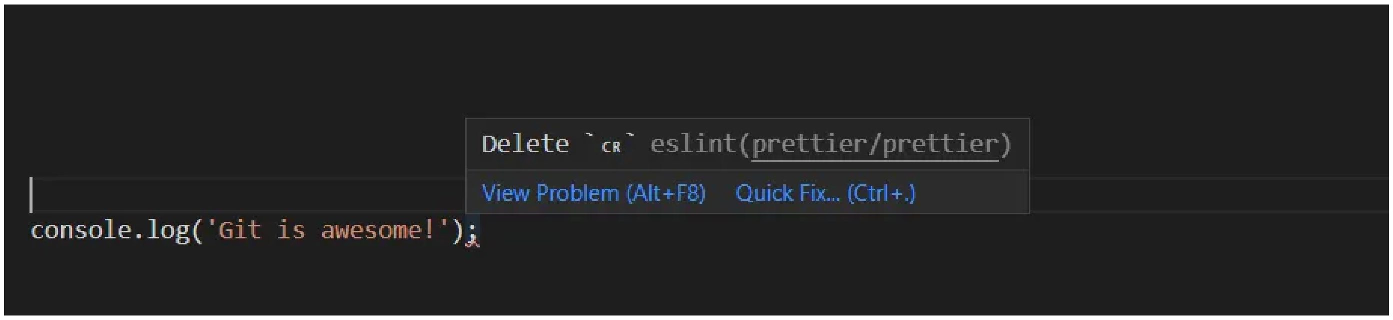
- `i` : The line endings in the index (what gets pushed to your repo). Should be `LF` .
- `w` : The line endings in the working tree (may be `CRLF` , but that's okay if the index is `LF`).
- `attr` : The `.gitattributes` rule that applies to this file.
- The file name itself.

Alternatively, you can double-check that git normalized your line endings correctly by re-cloning your repository on a Windows machine after you've pushed your code to the remote. You should see that both the index and working-tree copies of your files are

Bonus: Create an `.editorconfig` File

A `.gitattributes` file is technically all that you need in order to enforce the line endings that show up on the remote copy of your codebase. However, as we saw above, you may still see `CRLF` line endings locally for files that you created because `.gitattributes` doesn't tell git to change the working copies of your files.

Again, this doesn't mean that git's normalization process isn't working; it's just the expected behavior. However, this can get annoying if you're also linting your code with ESLint and Prettier, in which case they'll constantly throw errors and tell you to delete those extra `CR` s:



```
Delete `CR` eslint(prettier/prettier)
console.log('Git is awesome!');
```

Fortunately, you can take things a step further with an `.editorconfig` file; this is an editor-agnostic project that aims to create a standardized format for customizing the behavior of any given text editor. Lots of text editors (including VS Code) support and automatically read this file if it's present. You can put something like this in the root of your workspace:

Filename: .editorconfig

```
root = true

[*]
end_of_line = lf
```

In addition to a bunch of other settings, you can specify the line ending that should be used for any new files created through this text editor. That way, if you're on Windows using VS Code and you create a new file, you'll always see line endings as `LF` in your working tree. Linters are happy, and so is everyone on your team!

Summary

That was a lot to take in, but hopefully you now have a better understanding of the whole CRLF vs. LF debate and why this causes so many problems for teams that use a mixture of Windows and other operating systems. Whereas Windows follows the original convention of a carriage return plus a line feed (`CRLF`) for line endings, operating systems like Linux and Mac use only the line feed (`LF`) character. The history of these two control characters dates back to the era of the typewriter. While this tends to cause problems with software like git, you can specify settings at the repo level with a `.gitattributes` file to normalize your line endings regardless of what operating systems your developers are using. You can also optionally specify an `.editorconfig` file to ensure that new files are always created with `LF` line endings, even on Windows.

Attributions

The photo used in this post's social media preview was taken by Katrin Hauf (Unsplash).

Comments (5)

Post comment

This comment system is powered by the GitHub Issues API. You can learn more about how I built it or post a comment over on GitHub, and it'll show up below once you reload this page.

gregorybleiker commented 2 months ago

You just saved me a lot of time, thank you!

DhrumilDave5 commented 3 months ago

Thank You so much, I like articles like this where there is details like the historical ones.

SKSSSX commented 5 months ago

For an engineer, this rivet is very important, without it, you can't build a building

chriswayg commented 7 months ago *Edited*

Cool about the history. I still used teletypewriters with a dial up phone modem in a computer lab.

ekortright commented 7 months ago

Very useful. Thank you.

Thanks for reading!

Here are some useful links before you go.