Advanced Configuration

While Starship is a versatile shell, sometimes you need to do more than edit `starship.toml` to get it to do certain things. This page details some of the more advanced configuration techniques used in starship.

WARNING

The configurations in this section are subject to change in future releases of Starship.

TransientPrompt in PowerShell

It is possible to replace the previous-printed prompt with a custom string. This is useful in cases where all the prompt information is not always needed. To enable this, run `Enable-TransientPrompt` in the shell session. To make it permanent, put this statement in your `$PROFILE`. Transience can be disabled on-the-fly with `Disable-TransientPrompt`.

By default, the left side of input gets replaced with `>`. To customize this, define a new function called `Invoke-Starship-TransientFunction`. For example, to display Starship's `character` module here, you would do

```
function Invoke-Starship-TransientFunction {
  &starship module character
}
```

```
Enable-TransientPrompt
```

TransientPrompt and TransientRightPrompt in Cmd

Clink allows you to replace the previous-printed prompt with custom strings. This is useful in cases where all the prompt information is not always needed. To enable this, run `clink set prompt.transient <value>` where <value> can be one of:

- `always` : always replace the previous prompt
- `same_dir` : replace the previous prompt only if the working directory is same
- `off` : do not replace the prompt (i.e. turn off transience)

You need to do this only once. Make the following changes to your `starship.lua` to customize what gets displayed on the left and on the right:

- By default, the left side of input gets replaced with `>` . To customize this, define a new function called `starship_transient_prompt_func` . This function receives the current prompt as a string that you can utilize. For example, to display Starship's `character` module here, you would do

```
function starship_transient_prompt_func(promp
  return io.popen("starship module character"
    .." --keymap="..rl.getvariable('keymap')
  ):read("*a")
end
```

- By default, the right side of input is empty. To customize this, define a new function called `starship_transient_rprompt_func`. This function receives the current prompt as a string that you can utilize. For example, to display the time at which the last command was started here, you would do

```
function starship_transient_rprompt_func(prom
  return io.popen("starship module time"):rea
end
load(io.popen('starship init cmd'):read("*a")
```

TransientPrompt and TransientRightPrompt in Fish

It is possible to replace the previous-printed prompt with a custom string. This is useful in cases where all the prompt information is not always needed. To enable this, run `enable_transience` in the shell session. To make it permanent, put this statement in your `~/.config/fish/config.fish`. Transience can be disabled on-the-fly with `disable_transience`.

Note that in case of Fish, the transient prompt is only printed if the commandline is non-empty, and syntactically correct.

- By default, the left side of input gets replaced with a bold-green `❯`. To customize this, define a new function called `starship_transient_prompt_func`. For example, to display Starship's `character` module here, you would do

```
end
starship init fish | source
enable_transience
```

- By default, the right side of input is empty. To customize this, define a new function called `starship_transient_rprompt_func`. For example, to display the time at which the last command was started here, you would do

```
function starship_transient_rprompt_func
  starship module time
end
starship init fish | source
enable_transience
```

Custom pre-prompt and pre-execution Commands in Cmd

Clink provides extremely flexible APIs to run pre-prompt and pre-exec commands in Cmd shell. It is fairly simple to use with Starship. Make the following changes to your `starship.lua` file as per your requirements:

- To run a custom function right before the prompt is drawn, define a new function called `starship_preprompt_user_func`. This function receives the current prompt as a string that you can utilize. For example, to draw a rocket before the prompt, you would do

```
end

load(io.popen('starship init cmd'):read("*a")
```

- To run a custom function right before a command is executed, define a new function called `starship_precmd_user_func`. This function receives the current commandline as a string that you can utilize. For example, to print the command that's about to be executed, you would do

```
function starship_precmd_user_func(line)
  print("Executing: "..line)
end

load(io.popen('starship init cmd'):read("*a")
```

Custom pre-prompt and pre-execution Commands in Bash

Bash does not have a formal preexec/precmd framework like most other shells. Because of this, it is difficult to provide fully customizable hooks in `bash`. However, Starship does give you limited ability to insert your own functions into the prompt-rendering procedure:

- To run a custom function right before the prompt is drawn, define a new function and then assign its name to `starship_precmd_user_func`. For example, to draw a rocket before the prompt, you would do

```
}
starship_precmd_user_func="blastoff"
```

- To run a custom function right before a command runs, you can use the `DEBUG` trap mechanism. However, you **must** trap the DEBUG signal *before* initializing Starship! Starship can preserve the value of the DEBUG trap, but if the trap is overwritten after starship starts up, some functionality will break.

```sh
function blastoff(){
    echo "🚀"
}
trap blastoff DEBUG     # Trap DEBUG *before*
set -o functrace
eval $(starship init bash)
set +o functrace
```

Custom pre-prompt and pre-execution Commands in PowerShell

PowerShell does not have a formal preexec/precmd framework like most other shells. Because of this, it is difficult to provide fully customizable hooks in `powershell`. However, Starship does give you limited ability to insert your own functions into the prompt-rendering procedure:

Create a function named `Invoke-Starship-PreCommand`

```
  }
```

Change Window Title

Some shell prompts will automatically change the window title for you (e.g. to reflect your working directory). Fish even does it by default. Starship does not do this, but it's fairly straightforward to add this functionality to `bash`, `zsh`, `cmd` or `powershell`.

First, define a window title change function (identical in bash and zsh):

sh
```sh
function set_win_title(){
    echo -ne "\033]0; YOUR_WINDOW_TITLE_HERE 
}
```

You can use variables to customize this title (`$USER`, `$HOSTNAME`, and `$PWD` are popular choices).

In `bash`, set this function to be the precmd starship function:

sh
```sh
starship_precmd_user_func="set_win_title"
```

In `zsh`, add this to the `precmd_functions` array:

sh
```sh
precmd_functions+=(set_win_title)
```

If you like the result, add these lines to your shell configuration file (`~/.bashrc` or `~/.zshrc`) to make

For example, if you want to display your current directory in your terminal tab title, add the following snippet to your `~/.bashrc` or `~/.zshrc` :

```sh
function set_win_title(){
    echo -ne "\033]0; $(basename "$PWD") \007
}
starship_precmd_user_func="set_win_title"
```

For Cmd, you can change the window title using the `starship_preprompt_user_func` function.

```
function starship_preprompt_user_func(prompt)
  console.settitle(os.getenv('USERNAME').."@"
end

load(io.popen('starship init cmd'):read("*a")
```

You can also set a similar output with PowerShell by creating a function named `Invoke-Starship-PreCommand` .

```
# edit $PROFILE
function Invoke-Starship-PreCommand {
  $host.ui.RawUI.WindowTitle = "$env:USERNAME
}

Invoke-Expression (&starship init powershell)
```

Enable Right Prompt

Some shells support a right prompt which renders on the same line as the input. Starship can set the content supported in `right_format`. The `$all` variable will only contain modules not explicitly used in either `format` or `right_format`.

Note: The right prompt is a single line following the input location. To right align modules above the input line in a multi-line prompt, see the `fill` module.

`right_format` is currently supported for the following shells: elvish, fish, zsh, xonsh, cmd, nushell.

Example

```toml
# ~/.config/starship.toml

# A minimal left prompt
format = """$character"""

# move the rest of the prompt to the right
right_format = """$all"""
```

Produces a prompt like the following:

```
▶                                   starship
```

Continuation Prompt

Some shells support a continuation prompt along with the normal prompt. This prompt is rendered instead of the normal prompt when the user has entered an incomplete statement (such as a single left parenthesis or quote).

`'[∙](bright-black) '`.

Note: `continuation_prompt` should be set to a literal string without any variables.

Note: Continuation prompts are only available in the following shells:

- `bash`
- `zsh`
- `PowerShell`

Example

```
# ~/.config/starship.toml

# A continuation prompt that displays two fil
continuation_prompt = '▶▶ '
```

Style Strings

Style strings are a list of words, separated by whitespace. The words are not case sensitive (i.e. `bold` and `BoLd` are considered the same string). Each word can be one of the following:

- `bold`
- `italic`
- `underline`
- `dimmed`
- `inverted`
- `blink`
- `hidden`

- `fg:<color>`
- `<color>`
- `none`

where `<color>` is a color specifier (discussed below). `fg:<color>` and `<color>` currently do the same thing, though this may change in the future. `inverted` swaps the background and foreground colors. The order of words in the string does not matter.

The `none` token overrides all other tokens in a string if it is not part of a `bg:` specifier, so that e.g. `fg:red none fg:blue` will still create a string with no styling. `bg:none` sets the background to the default color so `fg:red bg:none` is equivalent to `red` or `fg:red` and `bg:green fg:red bg:none` is also equivalent to `fg:red` or `red`. It may become an error to use `none` in conjunction with other tokens in the future.

A color specifier can be one of the following:

- One of the standard terminal colors: `black`, `red`, `green`, `blue`, `yellow`, `purple`, `cyan`, `white`. You can optionally prefix these with `bright-` to get the bright version (e.g. `bright-white`).
- A `#` followed by a six-digit hexadecimal number. This specifies an RGB color hex code.
- A number between 0-255. This specifies an 8-bit ANSI Color Code.

If multiple colors are specified for foreground/background, the last one in the string will take priority.

quirks exist:

- Many terminals disable support for `blink` by default
- `hidden` is not supported on iTerm.
- `strikethrough` is not supported by the default macOS Terminal.app

Edit this page on GitHub

← Configuration Frequently Asked Questions →