vim-tene

vim-tene is a Vim9 script plugin. It produces a highly configurable, conceptually straightforward, statusline, using no user-defined functions, and is built via a sequence of expressions using ternary operators.

Contents

1. Example vim-tene statuslines

It is not easy to outline all of the features of vim-tene, though here are three condensed examples as a taster.

```
NORMAL  b1 ⊞ README.adoc       asciidoctor utf-8 unix ₙ193/1420 ₙ19 U+41,U+308
These are the default settings for the vim-tene's statusline¶
1 Buffer number¶
2 File name only¶
3 Line number and total lines¶
4 Virtual column¶
5 Unicode characters (up to three under the cursor: Ä (U+0041,U+0308) shown)¶
```

```
^V  <im-tene\README.adoc asciidoctor utf-8 unix 13% ×18-25 asciidoctorH1 ^V S
Other settings, which are not defaults:¶
1 mode(1) code instead of a full mode name¶
2 full path (not shown well here, truncated)¶
3 percent through the buffer¶
4 col() and {-num}¶
5 highlight group                                6 mode(1) and state()¶
```

```
REPLACE  P [+] ^K S @z README.adoc    asciidoctor utf-8 unix 13% ¶193/1420 |25
Configurable indicators. Powerline enabled font characters or ASCII (shown)¶
1 Paste               Not demonstrated:¶
2 Modified            * Preview¶
3 Digraph             * Encrypted¶
4 Spell               * Unmodifiable¶
5 Recording           * Read only¶
```

2. Installation

vim-tene is built primarily for Vim 9.

However, some of the later patches of Vim 8.2 work too. *See* Accommodating Vim 8 for details.

> ⚠️
>
> 1. Versions from 8.2.3555 through to the final patch of version 8.2[1] work almost as well as Vim 9.[2] However, since Vim9 script was not officially released until Vim 9 (2022-06-29), it is likely some of those later patched Vim 8.2 instances will not work as well as, especially a later, Vim 9 version.[3]
>
> 2. Versions from 8.2.3434[4] to v8.2.3554 mostly work, though not all modes will show in the statusline, e.g., Operator-pending does not trigger ModeChanged, which patch 8.2.3555 addressed.
>
> 3. Using Vim 8 versions earlier than 8.2.3434 (i.e., versions prior to the addition of the ModeChanged autocommand event, vim-tene will only produce a basic, static statusline.
>
> 4. It will not work with Neovim. That is not only because the code is Vim9 script, but also for reasons including that Vim's builtin function state(), which is used in several places, is not available in Neovim (at the time of writing, 2023-04-16).[5]

The recommended way to use this plugin is to use `packadd!` in your ~/.vimrc, though several options are outlined, below. If you use a plugin manager (other than vim-plug, the only manager included, below) it should have detailed instructions for how to install plugins using it.

2.1. Using packadd! in your .vimrc [recommended]

Either `git clone https://github.com/kennypete/vim-tene ~/.vim/pack/plugins/opt/vim-tene` *or* download the .zip from https://github.com/kennypete/vim-tene and unzip the contents within the folder vim-tene-main to ~/.vim/pack/plugins/opt/vim-tene.

You should have a directory structure like this (Linux and Windows respectively):

In your ~/.vimrc add `packadd! vim-tene`. For an example of this, *see* Using your .vimrc to packadd! vim-tene.

> 💡
>
> This is a contemporary and versatile way of using plugins. If you want to turn them off, it's easy — comment out `packadd! vim-tene`.

2.2. Vim's packages method, automatically

Similar to the above, except using `start` rather than `opt` …
Either `git clone https://github.com/kennypete/vim-tene ~/.vim/pack/plugins/start/vim-tene`
or download the .zip from https://github.com/kennypete/vim-tene and unzip the contents within the folder vim-tene-main to `~/.vim/pack/plugins/start/vim-tene`.

> This is a contemporary, though less versatile way of using plugins.
> If you want to turn one/more off you need to move it/them out of your `start` directory.

2.3. Using a plugin manager

For example, vim-plug (using "shorthand notation").

In the vim-plug section of your `.vimrc`, add `Plug 'kennypete/vim-tene'` between `call plug#begin()` and `call plug#end()`. Reload your `.vimrc` and `:PlugInstall` to install plugins.

> This is just one plugin example.
> Also, it is not the only way vim-plug could be used.

2.4. As a "global" plugin

As vim-tene is a unitary Vim9 script, it can be added easily as a global plugin to your `~/.vim/plugin` directory.

Download the .zip from https://github.com/kennypete/vim-tene and unzip the contents.

Copy `tene.vim` to your `~/.vim/plugin` directory.

If you want the help file too, copy `tene.txt` from the .zip file to your `~/.vim/doc` directory. Either copy `tags.` there too or rebuild the tags. That is explained in add-local-help.

> This may be your preferred method if you use very few plugins.
> It also avoids using contemporary means of using plugins.

2.5. Sourced directly by your ~/.vimrc

Finally, another option is to :source tene.vim directly or use :runtime.

Put tene.vim into your `~/.vim` directory (or anywhere in your runtime path) and `:ru tene.vim`.

You would not automatically get the benefits of the help file, but this just illustrates the simplicity, versatility, and portability that comes with being one .vim file.

> This may be your preferred method if you use a lot of different versions of Vim, including some really old ones.
> (Including because `packadd` will not work below version 704 with patch 1485).

1. If your operating system is Windows, instead of `~/.vim/`:

   - In PowerShell, use `$HOME\vimfiles\` or `~/vimfiles/`, or

   - In Command Prompt, use `%USERPROFILE%\vimfiles\`.

2. In the paths above, `plugins` may be whatever name you like (noting `'packpath'` is scanned for plugins under the `start` directory (automatically) and `opt` when `packadd` is executed).

3. You may also add configuration options to your `~/.vimrc` — *see* Configuration.

3. Aims

The vim-tene plugin began as an experiment to see whether a statusline with lots of features could work using only ternary[6] expressions (plus *with* Vim's builtin functions, but especially *without* complex user-defined functions, often spread across many vimscript files).

Aims expanded, as things progressed, to include:

1. Handle every mode, where practicable. Some statusline plugins don't display Vim Ex or Ex, for example. Operator-pending modes (no, nov, noV and noCTRL-V) also seemed to be either non-handled or ignored by statusline plugins.

2. Provide lots of configuration options, both at startup and interactively. The latter is important because not all editing scenarios, including intra-session, are the same. So, whereas knowing what Unicode character (or *characters*, i.e., including composing characters) is under the cursor may be critical sometimes, other times it may be inconsequential. Hence, providing interactive toggling of features was important.[7] Illustrating toggling variables shows this in action.

3. "Do no harm" / don't break Vim's core features.

4. Respect users' settings and colorschemes. In terms of the latter, keep it low effort by "recycling" default highlight groups, and leave it to users to do their own thing otherwise.

5. Keep things clean. Tab / triangular characters such as U+E0B0 and U+E0B1[8] provided by Powerline enabled / patched fonts consume statusline "real estate". Don't, by default, use those where there isn't any useful information provided to the user. So, line and column number indicators are fine (because they are no more verbose than "L" or "C", for example) whereas the pointer / triangular characters noted are not. Also enable using nothing at all if that is your preference, e.g., "282/1270 96" with no line or column indicators at all. Similarly colours: providing mode indicators in different colours makes sense because visually it reminds you you are in a certain mode/mode group, but applying colours everywhere because you can? NB: If you want *Joseph and the Amazing Technicolour Dreamcoat*, this isn't the plugin for you. 😁

6. Notwithstanding aim #5, provide for Unicode and Powerline characters for indicators like `'modified'`, `'readonly'`, etc. That's consistent with simplicity and utility, e.g., one character such as U+F457 consumes less screen space, and resembles Vim's default, i.e., [+], which consumes three characters. And making it configurable (Glyphs: Unicode or ASCII) means it's easy, if you want to use ASCII characters exclusively, only to use those.

7. Be independent of, but also not break, other plugins. Trying to be all things to all people is unwise. Plus aim #3 and aim #4 would not be met.

8. Run vim-tene in the way you prefer. Use `packadd!` manually, load with Vim's packages automatically, load with a plugin manager, or `:source` / `:runtime` vim-tene.vim directly. Make all those

options straightforward.

9. Use Vim9 script. Vim 9 (or Vim 8.2 with at least patch 3434, though preferably with patch 3555) is needed because the ModeChanged autocommand event, is essential — *see* Autocommand group — and since Vim9 script was enabled at that point it was feasible to use it.[9]

10. Provide a static statusline that's more feature rich than the standard when sourced with a version that cannot handle Vim9 script — *see* Accommodating Vim 8 — so that vim-tene does not cause errors when sourced by older versions.

11. Don't utilise any user-defined functions.

12. Utilise only (a) expressions using ternary operators, and (b) Vim's builtin functions.

Keeping to these aims mostly reduced complexity.[10] Some statusline plugins have extraordinary amounts of code, often dedicated to accommodating other plugins, appearing to prioritise æsthetics over utility.

3.1. The standard statusline

When Vim loads, and 'laststatus' equals 2, Vim will draw a statusline at the bottom of each window. Vim's "standard" statusline may be emulated with one short command:

```
set statusline=%<%f\ %h%m%r%=%-14.(%l,%c%V%)\ %P
```

A "translation" of that command is, "Set the statusline to":

1. Truncate the statusline at the start, when necessary (%<)

2. Show the path to the file in the buffer, as typed or relative (%f)

3. Insert a space (\ )

4. Display "[help]" buffer flag, when applicable (%h)

5. Display "[+]" 'modified' flag / "[-]" if 'modifiable' is off (%m)

6. Display "[RO]" 'readonly' flag, when applicable (%r)

7. Right align the remainder of the 'statusline' (%=)

8. Pad with spaces up to 14 characters (%-14.(…))

9. Display line number, a comma, and column-virtual column (%l,%c%V)

10. Insert a space (\ )

11. Display All/Top/Bot/percentage through the buffer's window (%P)

To illustrate, the "standard" statusline, for a modified, unsaved buffer, and with no content will appear like this:

```
[No Name] [+]                                          1,0-1          All
```

This plugin, as others do, extends way beyond the "standard". In this plugin's case, there are myriad options set by default. Many may be configured (*see* Configuration) in your ~/.vimrc or via <Leader> mappings, which may be used to toggle features interactively (*see* Leader mappings).

Although a great deal is configurable, there are limits. For example, unlike some statusline plugins, the order of what appears is fixed. That is a pragmatic limitation of the self-imposed constraints noted in Aims and Features, including only using ternary expressions. Whether that is a limitation you are willing to accept is up to you.

Another limitation is colours (highlight groups): they have been set to leverage a few of Vim's default highlight groups. That's because those won't be reset when a colorscheme is loaded and highlight-clear is executed. It has also been limited to the mode indicator/name and "the rest". That is a sensible decision because it would make the ternary expressions extremely complicated if lots of highlighting optionality was included (and that's unnecessary). It is also consistent with keeping things "clean" — *see* aim #5.

> The 'tabline' is not in scope of this plugin. It is **statusline only**. The default tab handling in Vim is fine (if tabs are used as described in the tabpage introduction, not proxy buffers[11]).
>
> Here's a gvim option for your ~/.vimrc or ~/.gvimrc if you want a 'tabline' with information that *is* useful (buffer numbers appearing in each tab and the tab's active window's buffer name):

```
set guitablabel=%{%join(tabpagebuflist('%'),'\ ◆\ ')..'\ %t'%}
```

4. Features

✓ Handles *all* of Vim's modes,[12] except modes where no 'statusline' is displayed — *see* All of Vim's modes demonstrated

✓ Shows pending states, i.e.:

  • Operator-pending (no, plus nov, noV and noCTRL-V), and
  • In Insert (i.e., including Replace and Virtual Replace) modes:
      ◦ CTRL-K (digraphs),
      ◦ CTRL-R (registers), and
      ◦ CTRL-V (special keys or decimal, octal or hexadecimal values of a character — i_CTRL-V_digit)

✓ Up to three characters under the cursor may be identified, i.e. with up to two composing characters identified — so, up to three Unicode U+nnnnn can be shown, e.g.:

  • **A** will show U+61
  • **Ā** will show U+61,U+304
  • **Ą̄** will show U+61,U+328,U+304

    (*See* Unicode character identification.)

✓ Many configuration options to turn on/off features and change information

✓ Configuration via ~/.vimrc and interactively with remappable <Leader> mappings

✓ Independent — neither impacts other plugins nor relies on any

✓ Only uses ternary expressions and Vim's builtin functions, and no user-defined functions

✓ Unitary — it's a single Vim9 script

✓ Sequential — the Vim9 script may be read line-by-line

✓ Terse — it's only around 140 lines of substantive code, albeit some are very long!

✓ Fast[13] - consistently sourced in only 2ms-4ms.[14]

4.1. All of Vim's modes demonstrated

Demonstration of modes

5. Configuration

There are many configuration options:

- Mode Names
- Digraph/Register/Special character "pending"
- Glyphs: Unicode or ASCII
- Binary variables for toggling features
- Leader keys

Highlight groups are discussed separately. *See* Highlighting.

5.1. Mode Names

Default names are set for modes so if you are happy with those there is nothing to do. If you do want to change a name (or two, or all) it is simply a matter of adding a few lines of code to your ~/.vimrc; the following example illustrates changing NORMAL to MĀORI, which in Māori literally means normal!

```
let g:tene_modes = exists("g:tene_modes") ? g:tene_modes : {} ①
let g:tene_modes["n"] = "MĀORI" ②
```

① adds the empty dictionary if it does not already exist, but leaves the dictionary as-is if it does).

② illustrates configuring the text that will be displayed for key "n".

The configurable items are listed in the table below. Renaming them from the default (column 2) involves determining the right dictionary key from column 1, e.g., in the example above, "n", and choosing whatever you want the mode name to appear as.

g:tene_modes['…']	Mode Name (default)	From Normal Mode
n	NORMAL	
no	OP PENDING	d
nov	OP PENDING (v)	d v

g:tene_modes['...']	Mode Name (default)	From Normal Mode
noV	OP PENDING (V)	`d``V`
noCTRL-V	OP PENDING (^V)	`d``CTRL-V`
niI	INSERT (NORMAL)	`i``CTRL-O`
niR	REPLACE (NORMAL)	`R``CTRL-O`
niV	VIRTUAL REPLACE (NORMAL)	`g``R``CTRL-O`
nt	TERMINAL (NORMAL)	`:term``CTRL-W``N`
v	VISUAL	`v`
vs	SELECT (VISUAL)	`g``h``CTRL-O`
V	VISUAL LINE	`V`
Vs	SELECT (VISUAL LINE)	`g``H``CTRL-O`
CTRL-V	VISUAL BLOCK	`CTRL-V`
CTRL-Vs	SELECT (VISUAL BLOCK)	`g``CTRL-H``CTRL-O`
s	SELECT	`g``h`
S	SELECT LINE	`g``H`
CTRL-S	SELECT BLOCK	`g``CTRL-H`
i	INSERT	`i`
ic	INSERT COMPLETION C	`i``CTRL-X``CTRL-]`
ix	INSERT COMPLETION X	`i``CTRL-X`
R	REPLACE	`R`
Rc	REPLACE COMPLETION C	`R``CTRL-X``CTRL-]`
Rx	REPLACE COMPLETION X	`R``CTRL-X`
Rv	VIRTUAL REPLACE	`g``R`
Rvc	VIRTUAL REPLACE COMPLETION C	`g``R``CTRL-X``CTRL-]`
Rvx	VIRTUAL REPLACE COMPLETION X	`g``R``CTRL-X`
c	COMMAND-LINE	`:`
cv	VIM EX	`g``Q`
ce	EX	`Q`
t	TERMINAL-JOB	`:term`

1. The keys are identical to the mode codes used in mode().

2. In column 3, these are illustrative only, e.g., d V is not the only way to get to linewise Operator-pending (noV) mode. (NB: Spaces are included for readability only.)

3. Modes r, rm, r? and ! have no statusline, so are not defined. The modes above are

listed in the order in mode() (and which is followed by vim-tene). The defaults are built into the ternary expressions using get(), i.e.,

```
get({dict}, {key} [, {default}])
```

This is an efficient way to set defaults for the 31 names.

5.1.1. Vim's 'showmode' option

Vim's default approach is to 'showmode', which puts a message on the last line, for example, when in Insert mode, -- **INSERT** -- is displayed.

Some Vim users turn off 'showmode' when there is a statusline plugin active. You may choose to do that, of course. However, there are times when the combination of 'showmode' and a statusline mode indicator are really useful. An example is where CTRL-O is used in Insert mode and v is entered. The mode, as revealed by mode/state indicators, is v (Visual) and the state is S (not triggering SafeState or SafeStateAgain) so the mode indicated in the statusline should be v or VISUAL (i.e., if defaults are used: mode(1)==#"v"). However, 'showmode' will display -- **(insert) VISUAL** --, which is more precise because you are not simply in Visual mode (accessed, via Normal mode, entering v). The critical point is, you will revert to Insert mode when you leave Visual mode. So, turn off 'showmode' if you wish, hiding Vim's default information, but only if you accept such downsides.

5.2. Digraph/Register/Special character "pending"

S state occurs in an Insert mode, i.e., any of Insert, Replace (R) or Virtual Replace (Rv), when you type one of:

CTRL-K

Enters a digraph, e.g., from Insert mode, `CTRL-K o o` produces U+2022, a bullet (•)

CTRL-R

Inserts the contents of a register, e.g., `:` puts the most recent command-line command

CTRL-V

Either inserts literal characters, e.g., a `Tab` *even when that is usually overridden with* 'softtabstop', or, e.g., a Unicode character such as `CTRL-V u 2 0 2 2` will enter the • character.

> `CTRL-Q` is a synonym for `CTRL-V`, which is useful if you ever find MS Windows preventing you using `CTRL-V` (commonly used by Windows for "paste").

The default indicator has been set to " I ". That may be overridden by setting g:tene_state_S in your ~/.vimrc. For example, if you wanted something ludicrously verbose:

```
let g:tene_state_S = ' iK/iR/iQ/iV Pending '
```

If you do not want anything to appear, no problem, `let g:tene_state_S = ''` will do that.

5.3. Glyphs: Unicode or ASCII

Several indicators may appear in a statusline. Some common ones are **[+]**, which indicates a modified buffer, and **[help]**, which shows you that the buffer is of the type Vim help. These are part of Vim's "standard" statusline, discussed in The standard statusline.

Many other indicators could be displayed. Some are useful, such as when a 'key' is encrypting the file you're editing. Vim has masses of options, some of which make sense to display when they're set, others not so much.

The default is to show symbols/glyphs, which include a few Powerline characters. Whether that's the right default is debatable, though users who prefer it off are probably more capable generally, so adding the line to make that happen should be a breeze for them, i.e.:

```
let g:tene_glyphs = 0
```

When this variable is set to 1, the default glyphs are ones that display nicely with font FiraCode NFM. Some are Powerline characters such U+30A1 used to indicate the line number. When set to 0, the ASCII character used for line number is the underscore (_), which is ASCII 95 (U+005F).

Vim-tene sets ASCII character and glyph defaults, so, if you are happy with the default ASCII and/or Unicode glyphs, there is nothing for you to do. If you do want to change one (or two, or all) it is simply a matter of adding the applicable lines of code to your ~/.vimrc. Illustrated below, is changing the digraph indicator to the &#e6; ligature (U+00E6) when ASCII and, when Unicode, ǽ (U+01FD). The former will be used when g:tene_glyphs is 0 and the latter when it is 1.

```
let g:tene_ga = exists("g:tene_ga") ? g:tene_ga : {}
let g:tene_ga["dg"] = ['æ', 'ǽ']
```

> **ⓘ** The code above overrides the following command in vim-tene, which, but for the above, defaults the digraph indicators to ^K (ASCII 94 and 75) and Æ (U+00C6) for Unicode.
>
> ```
> g:tene_ga['dg'] = has_key(g:tene_ga, 'dg') ? g:tene_ga['dg'] :
> ['^K', 'Æ']
> ```

Another example: this time, the line number indicator. If you wanted the pilcrow rather than underscore, regardless of the value of g:tene_glyphs, you'd use:

```
let g:tene_ga["line()"] = ['¶', '¶']
```

All of g:tene_ga dictionary's configurable items, and their ASCII and Unicode glyph defaults, are shown below.

```
Dictionary item                     ASCII            Glyph   Unicode  Scope
---------------                     -----            -----   -------  -----
g:tene_ga['buftypehelp']            [help]            ?      U+F128   local
g:tene_ga['paste']                  P                 P      U+F8E2   global
g:tene_ga['mod']                    [+]               ⊞      U+F457   global
g:tene_ga['noma']                   [-]               ⊟      U+F458   global
g:tene_ga['pvw']                    [Preview]         ▣      U+F1C4   local
g:tene_ga['dg']                     ^K                Æ      U+00C6   global
g:tene_ga['key']                    E                 ⚿      U+F80A   local
g:tene_ga['spell']                  S                 ✓      U+F42E   local
g:tene_ga['recording']              @                 ⊙      U+F519    -
g:tene_ga['ro']                     [RO]              ⦸      U+F05E   local
g:tene_ga['line()']                 _                 ᴸₙ     U+E0A1   local
g:tene_ga['col()']                  c                 ⤬      U+EAF5   local
g:tene_ga['virtcol()']              |                 ᶜₙ     U+E0A3   local
```

If there are any you do not want to display at all, e.g., if you wanted line numbers but never any indicators, just do this:

```
let g:tene_ga["line()"] = ['', '']
```

Some of these variables may be self-evident, though it is worth explaining what each is doing, nonetheless.

g:tene_ga['...']	Displays ASCII/glyph when?
'buftypehelp'	The buffer is of type help.
'paste'	Option &paste is on (Vim is in "Paste mode").
'mod'	The buffer has been modified.
'noma'	The buffer is not modifiable.
'pvw'	The buffer is a preview window.
'dg'	When set, the second method for entering digraphs (i.e., character-backspace-character) e.g., a `<BS>` e to enter the ligature æ, is enabled. The only modes where entering a digraph in that manner is allowed are Insert, Replace, Virtual Replace, Command-line, and Vim Ex (i.e., g Q), so only show the indicator if one of those modes is the current mode.
'key'	Display a key indicator when the 'key' option (i.e., the buffer is encrypted). And, if the 'cryptmethod' is not blowfish2, show that too. (Other types are discouraged).
'spell'	Display a spell-checking indicator.
'recording'	Display a macro recording indicator when one is being recorded as well as the register to which it is being recorded. It's more useful when showmode is off, but still useful to have it indicated in the statusline.
'ro'	Display a read only flag. %R or %r could be used, but this provides optionality (e.g., if you want to use a symbol for read only, which is what's been enabled).
'line()'	Display a line number indicator "prefix".
'col()'	Display a byte column indicator "prefix".
'virtcol()'	Display a virtual column indicator "prefix".[15]

5.3.1. Statusline options illustrated

The images below illustrate most of the ASCII and Unicode indicators as they appear on a vim-tene statusline (using the defaults):

ASCII defaults

Unicode defaults

5.4. Binary variables for toggling features

Several variables enable toggling of features. For example, g:tene_buffer_num, when set to 1, will display b{buffer number} after the mode indicator. The defaults are indicated in the table below. They may be overridden by setting them to the opposite in your ~/.vimrc (i.e., 0 if the default is 1, and vice versa).

Variable	Default	Shows ...
g:tene_buffer_num	**1**	b{buffer number} after mode indicator
g:tene_file_tail		file name only, not the full path
g:tene_glyphs		*See* Glyphs: Unicode or ASCII
g:tene_keymap		display <b:keymap_name> in mode label
g:tene_line_num		line number (cursor position)
g:tene_line_nums		total number of lines in the buffer
g:tene_virtcol		virtual column number, virtcol()
g:tene_unicode		U+nnnnn of character(s) at the cursor
g:tene_col	**0**	col() (and -{num} if %V is different)
g:tene_hl_group		highlight group under the cursor
g:tene_mode		n, ce instead of NORMAL, EX, etc.
g:tene_modestate		mode(1) and state() codes, e.g., i S
g:tene_path		full filepath of the buffer
g:tene_percent		% (at the cursor) through the buffer

Setting these will be a matter of preference / how you use Vim. For example, some users have little interest in buffer numbers so may wish to `let g:tene_buffer_num=0` whereas other users may use buffers a lot and find the default, showing buffer numbers, essential, even for aiding doing things like `:[N]sb` — *refer* sb (which splits the current window and edits buffer [N]).

Some toggles will be more useful depending on the editing scenario. That is why the ability to not only set them in your ~/.vimrc, but also toggling interactively has been enabled.

5.4.1. Mode and state indicator

For example, while creating this plugin it was priceless having `g:tene_modestate`, which shows not only the current mode(1) code but also the state(). For example, when in Normal mode and d is pressed, the mode and state are "no oxS", with "no" meaning mode Operator-pending and state "oxS".

The o, x, and S mean:

o	operator pending
x	executing an autocommand
S	not triggering SafeState or SafeStateAgain

5.4.2. Keymap

Some of the things that can be toggled may be inconsequential to some users, with keymaps ('keymap'), being an example. Although it has been made active, many users will never see its manifestation in the statusline, which is set to display the value of b:keymap_name (*aka* %k) when it is set, e.g., such as when:

```
let &keymap="german-qwertz"
```

This will change mode indicators like <mark>INSERT</mark> to <mark>INSERT <de></mark> (for all the Insert modes — i, R, and Rv — and Command-line mode). For anyone who does not use keymaps, this is of little importance. But, for those who do, it may be preferable to know that the 'keymap', is active. If you really do not want it at all, `let g:tene_keymap=0` in your ~/.vimrc will do that.

5.4.3. Unicode character identification

Unicode character identification using `g:tene_unicode=1` is worth explaining. Vim provides for %b and %B, which the help for statusline says will show:

```
%b   N    Value of character under cursor.
%B   N    Value of character under cursor, in hexadecimal.
```

The `%b` and `%B` items are fine, however, they only consider a *single* character under the cursor, not composing characters.

> The `%B` item may be expressed as:
>
> ```
> printf('%X',char2nr(matchstr(getline('.')[col('.')-1:-1],'.')))
> ```

Handling composing characters is a good idea though too. Examples are:

- **Ä** (U+0041,U+0308), *not to be confused with* **Ä** (U+00C4), which illustrates another benefit, and
- **Ą̈** (U+0041,U+0308,U+0327)

Refer https://unicode.org/faq/char_combmark.html

These have been handled so that up to three characters under the cursor will be shown in the statusline. The last example, above, displays **U+41,U+308,U+327**, as shown below.
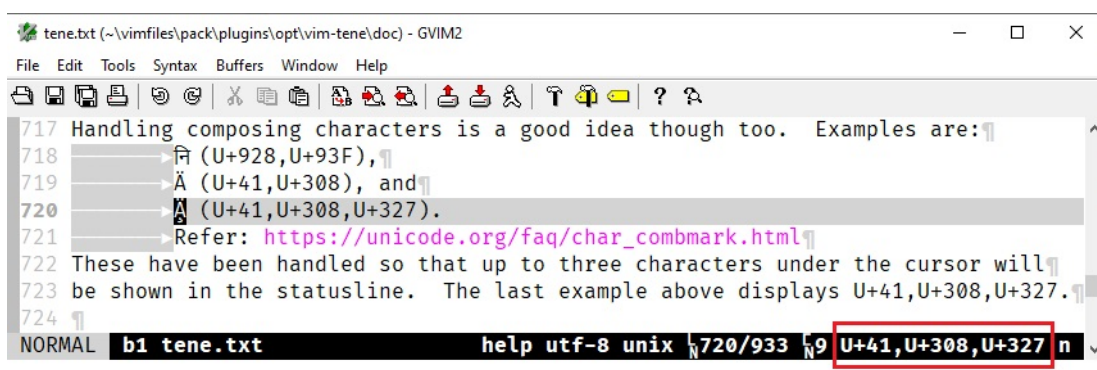
> Technically, there should be four digits following the U+. That has not been done because it frequently demands two additional zero characters without providing any additional information.

This is also helpful when using a font that includes programming ligatures, e.g., `!=` (U+21,U+3D) could be rendered as something that looks like a (two-character width) ≠ (U+2260). Putting aside the merits of programming ligatures (no thanks, IMHO), it is your choice: identifying, via the statusline, the Unicode characters in your buffers may be useful.

5.4.4. Illustrating toggling variables

Demonstration of toggling variables

5.5. Leader keys

As listed in Default <Leader> mappings, there are 14 <Leader> key mappings set by default. Those may be changed to different key mappings — as explained in Leader mappings, a default will not be set when there is a mapping you have made.

Say you do not like the mapping of <Leader>tz to <Plug>TeneZ for toggling line numbers and you wanted <Leader>tt to do that instead. In your ~/.vimrc, this would achieve that:

```
map <Leader>tt <Plug>TeneZ
```

This prevents the mapping of <Leader>tz, the default, to <Plug>TeneZ because <Plug>TeneZ is already mapped to <Leader>tt by the time vim-tene is sourced.

Another example: you prefer to have <Leader>tp toggle the percentage (through the file) indicator. This would achieve that:

```
map <Leader>tp <Plug>Tene%
```

Be aware though, this not only creates that overriding mapping but:

a. the default mapping of <Leader>t% now will not be mapped because `!hasmapto('<Plug>Tene%')` is now false (i.e., it is mapped), and

b. <Leader>tp will no longer be defaulted to <Plug>TeneP because maparg will determine that <Leader>tp has been mapped already (*refer* Leader mappings).

5.6. Colour Highlighting

Mode indicators, the active statuslines, and inactive statuslines' colours are all configurable — *see* Highlighting.

6. Autocommand

There is only one autocommand:

```
autocmd ModeChanged *:[^t]\+ redrawstatus
```

The ModeChanged autocommand event, enabled on 2021-09-13 (v8.2.3434), was an essential improvement, enabling easy statusline display of all modes (other than those which do not have statuslines, i.e., r, rm, r?, and !).

Redrawing the statusline when the mode has changed is essential in some cases, e.g., when entering Ex mode with `Q`, because without the ModeChanged autocommand event, it does not appear to be displayable. Compare other statusline plugins: do any *not* continue to display Normal when in Ex mode? Even more important than Ex mode and Vim Ex mode (i.e., `gQ`) is the Operator-pending mode. Without the ModeChanged autocommand event, displaying mode indicators for mode(1) codes no, nov (o_v), noV (o_V), and noCTRL-V (o_CTRL-V) is either impossible or not obviously achievable.[16]

The only exception that's been handled is when going to terminal mode.[17] There are some instances where 'showmode' displays incorrect information when redrawstatus is executed upon entering a terminal window.[18] Consequently, the autocommand for the ModeChanged autocommand event excludes redrawing the statusline when entering a terminal window.

7. Highlighting

Highlighting has been kept simple. This has been achieved by leveraging five of Vim's non-statusline default highlight groups (DiffAdd, ErrorMsg, Pmenu, Visual, and WildMenu),[19] only applying distinct highlighting to mode indicators, and leaving the rest to four of Vim's default highlight groups, i.e., StatusLine, StatusLineNC, StatusLineTerm, and StatusLineTermNC.

Two benefits in doing this include:

- Little time is wasted messing with colours and, critically,

- Vim's default highlight groups are immune from being cleared by highlight-clear, which colorschemes normally execute.

The dictionary `g:tene_hi` (which, by default, is empty) has up to nine items that may be used to configure highlight groups to your liking. By default, i.e., if none are overridden, the following groups are used:

g:tene_hi['…']	Highlight group	Used for
c	StatusLineTermNC	Command-line and Ex modes' indicators, and the inactive terminal statusline (the entire line)
i	WildMenu	Insert mode indicator
n	Visual	Normal (n, plus niI, niR, and niV via i_CTRL-O) and Terminal-normal (nt) modes' indicators
o	ErrorMsg	Operator-pending modes (no, plus nov, noV and noCTRL-V) indicators and " I " in S state — *see* g:tene_state_S)

g:tene_hi['…']	Highlight group	Used for
r	Pmenu	Replace mode indicator
s	StatusLine	Active statusline after the mode indicator
t	StatusLineTerm	Active terminal statusline after the t or nt mode indicator
v	DiffAdd	Visual and Select modes' indicators
x	StatusLineNC	Inactive statuslines (the entire statusline)

Since the default highlight groups differ depending on the background, using this approach is also dynamic in that, if the colorscheme or background changes, the statuslines do too.

Of course, these highlight groups may not be to your liking. To change any of them, add to your ~/.vimrc, before where vim-tene is loaded, the following:

```
let g:tene_hi = {}
```

Then specify whatever overrides you want for any of the nine items listed above. For example, if you use Windows gvim, the default light scheme has an inactive statusbar that is the same background colour as the active one. To make it more obvious that it's inactive, the following could be added:

```
let g:tene_hi['x'] = 'Conceal'
```

In default Windows gvim this is guifg=LightGrey and guibg=DarkGrey.

7.1. Demonstration of colorschemes

Demonstration of colorschemes

7.2. Using an augroup

Whether you love it or loathe it, the gruvbox colorscheme seems to be many Vim users' favourite colorscheme. If in the "love it" camp, keep reading; if in "loathe it", skip to A mode behaviour (to avoid?).

With gruvbox as your colorscheme — tested in Windows gvim only — one approach is to add an augroup to your ~/.vimrc to have colours that are more æsthetically in keeping with that colorscheme:

```
augroup gruvbox_tene
  autocmd!
  autocmd ColorScheme gruvbox {
    g:tene_hi = exists("g:tene_hi") ? g:tene_hi : {}
    g:tene_hi['o'] = 'DiffDelete'
    g:tene_hi['i'] = 'IncSearch'
    g:tene_hi['r'] = 'DiffText'
    g:tene_hi['v'] = 'DiffChange'
  }
augroup END
```

Of course, this is only illustrative. You are not limited to re-using the default highlight groups. So, you could define your own, adding to the above augroup something like:

```
hi tene_x gui=italic guifg=#dadada guibg=#d5c4a1
g:tene_hi['x'] = 'tene-x'
```

… and "*da da*", the Inactive statuslines ('x') will now appear with a gross italic grey on a sickening pastel tan background. 😭

8. A mode behaviour (to avoid?)

This final section is not about vim-tene specifically - it's about behaviour identified when writing the plugin. That's the retention of Insert mode when entering the window of an unmodifiable, buffer.[20]

Initially (in development) a couple of BufEnter autocommands were used to address the default behaviour whereby, if you are in one of the Insert modes (Insert, Replace or Virtual Replace) and click (or CTRL-O CTRL-W w) into an unmodifiable buffer, e.g., a netrw or help buffer, the applicable Insert mode persists, despite the buffer not allowing changes. There are scenarios where this may be wanted, e.g., if you are transiting through windows and do not want the Insert mode to change, though it seems unlikely that is what most users would want/expect. (And it fails when there's a terminal window somewhere in the transit because entering a terminal window stops the Insert mode). The downside of retaining the Insert mode is that entering a netrw or help buffer, will generate an error upon pressing almost any key aside from arrow keys.

For anyone who does not want that behaviour, vimscript like the following may be added to your ~/.vimrc:

```
augroup forcenormal
  autocmd!
  autocmd BufEnter * execute (!&modifiable && !&insertmode)
        \ ? ':call feedkeys("\<Esc>")' : ''
  autocmd BufEnter * execute (!&modifiable && &insertmode)
        \ ? ':call feedkeys("\<C-L>")' : ''
augroup END
```

This sends the applicable keys to the unmodifiable buffer's window when it's entered, changing it to Normal mode. There are two things to be aware of if you opt for this. First, there may be a bell. Adding :set belloff+=esc, to your ~/.vimrc, is the solution to avoid that annoyance. Second, if you have a navigation mapping, e.g., inoremap <F3> <C-O><C-W>w, it will be "broken" insofar as if you navigate to a window where it's been changed to Normal, the <F3> will no longer be applicable because Insert mode will have ended.

The augroup code, above, also handles 'insertmode'. When that option is set, it makes Vim work in a way that treats Insert mode as the default mode. A consequence of having Insert as the default is that it applies when entering netrw or other unmodifiable buffers' windows. The second BufEnter command sends CTRL-L, making the unmodifiable buffer automatically go to Normal mode when entered (specifically when 'insertmode' is set).

9. Code "walk through"

This walk through provides details and, in places, reasons why things were done the way they were. It's feasible because tene.vim is only ~140 lines of Vim9 script (excluding commented lines). It is not a detailed run through of the configuration options or mappings. (For those, *see* to Configuration and Leader mappings.)

9.1. Accommodating Vim 8

Although vim-tene is designed to work with Vim 9 (or Vim8 from 8.2.3434 - *see* Installation), prior to the Vim9 namespace is some vimscript. It tests for whether the version of Vim is neither 9 nor 8.2 with patch 3434. If neither are so, it sets the statusline to something more feature rich than The standard statusline. This exploits the limited ability to precede Vim9 script with vimscript — *refer* vim9-mix.

If sourced with Vim 8.2 (with patch 1705), a warning may be provided in a popup like this:
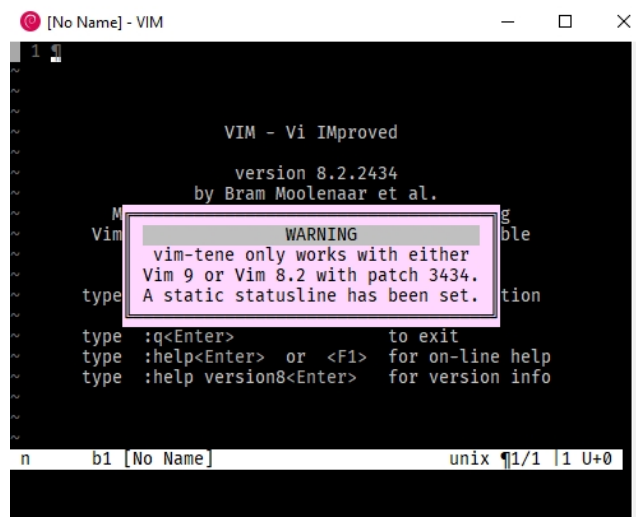
This may be enabled by defining the variable `g:tene_nowarn` in your ~/.vimrc, noting it can be defined as *anything*. You may want that when you use Vim 9 most of the time and a Vim 8.2 version, without patch 3434, only occasionally.

Versions before 8.1 patch 1705 cannot show the popup warning — it errors — but the static statusline is still applied. This has been tested, and works on versions 8.1, 8.0, 7.4, 7.3, and 7.2.[21] To illustrate, you may use gvim 9 in Windows, Vim 9 in PowerShell, Vim 8.2.2434 in Debian 11 stable (as shown above; NB: that is the version provided with Bullseye — Package: vim (2:8.2.2434-3+deb11u1)), Vim 8.2.4836 in iSH, and iVim 8.1 on an iPhone. In a scenario like that, the same ~/.vimrc could be used (bar the naming, i.e., _vimrc in Windows), using packadd! to source vim-tene, and letting vim-tene set the static statusline when the version is either <8.2 or is Vim 8.2 without patch 3434.

In summary, vim-tene will enable the following:

Version	Patch	vim-tene features enabled
9	*all*	All.
8.2	>=3555	
8.2	>=3434	All features, but excluding Operator-pending handling.

Version	Patch	vim-tene features enabled
8.1	>=1705	Static statusline with optional warning with `g:tene_8warn`.
8.x or 7.?	*all*	Static statusline with no optional warning.

9.1.1. Using your .vimrc to packadd! vim-tene

Vimscript in your ~/.vimrc could enable use of vim-tene regardless of whether the version is 7, 8, or 9, provide the popup warning (if available), and also handle the scenario where vim-tene itself either is unavailable or fails, for whatever reason. This is vimscript that could enable this:

```
try
  let g:tene_8warn = 1
  packadd! vim-tene
catch
  set statusline=\ %-5.(%{mode(1)}%)%<%t,b%n%M%R%H%Y%=%{&ff}\ %l/%L,%c%V\ %P
endtry
```

> This presumes you are using packadd!, not sourcing vim-tene with a plugin manager.

9.2. Namespace

The vim9script statement "tells Vim to interpret the script in its own vim9-namespace".

9.3. Dictionaries

There are three dictionaries. All are empty by default.

g:tene_hi{}

> You may use this to override default highlight group settings. Those are explained in Highlighting.

g:tene_modes{}

> You may use this to override default mode names, i.e., the names displayed at the start of the statusline. The defaults are listed in Mode Names, and include "INSERT", "VISUAL BLOCK", etc.

g:tene_ga{}

> You may use this to override default glyphs/symbols, which are used for indicators like 'key' (U+F80A), 'spell' (U+F42E), etc. The defaults are listed in Glyphs: Unicode or ASCII.

> The Unicode defaults have been tested with the Powerline enabled font FiraCode NFM in Windows gvim (8 and 9) and Vim 9, PowerShell Vim 9, iVim Vim 8, iSH Vim 8, gvim and Vim 8 in Debian 11, and Vim 8 WSL Debian 11. (NB: "8" here means 8.2 with patch 3434.) Setting `g:tene_symbols` to 0 avoids (by default) using characters other than ASCII ones, e.g., character K for 'key' and character S for 'spell' in the examples above.

9.4. Variables

9.4.1. g:tene_state_S

This is a special variable used to prepend (only before Insert modes' names, i.e., modes i, R, and Rv), an indicator that one of i_CTRL-K (digraphs), i_CTRL-R (registers), or i_CTRL-V/i_CTRL-Q (special characters) is "pending". When Vim awaits character input in those special cases, it goes into state S (SafeState). The default has been set to " I ", i.e. **space I space**, which provides a succinct indicator that Vim is awaiting input after CTRL-K (?), CTRL-R ("), or CTRL-V/CTRL-Q (^).

> Ideally, a similar indicator would be possible for other S state scenarios. For example, f, F, t, T, g, and [count], all await further input when used in Normal mode. However, although Vim's help says (in state()):
>
> ```
> S    not triggering SafeState or SafeStateAgain, e.g. after f or a count
> ```
>
> it does not seem to do so after f (or the other S state scenarios); that is, it doesn't appear to be detectable using the state() builtin function.

9.4.2. Variables that may be toggled

These variables provide for options that may be turned on/off. For example, g:tene_glyphs defaults to 1, so Unicode characters (outside of ASCII's range, including some Powerline ones) are used for indicators. If set to 0, in your ~/.vimrc, only ASCII characters will be used (by default, i.e., you may change those if you want to). There is an option to toggle interactively with <Leader>tg too. These options are explained in detail in Binary variables for toggling features.

9.5. Autocommand group

The tene augroup starts with autocmd!, which clears the existing autocommand if the plugin is sourced when already loaded to ensure it does not appear twice.

The sole autocommand is critical, i.e., ModeChanged. Without this, some modes (examples: Ex mode and Operator-pending) are not detected, or at least there appears no obvious way to detect them. From using a few other statusline plugins, it seems either they couldn't detect such modes or some modes that were not handled because they are uncommon (like Ex mode), so nobody contemplated handling them?

When a change in mode is detected, redrawstatus is used, which ensures the active statusline is redrawn. That may be in one of the modes that is normally not easily (or not at all?) detectable. Those include Operator-pending modes, with modes no, nov, noV, and noCTRL-V all indicated when applicable (such as when pressing d in Normal mode (mode no), and then v (mode nov), V (mode noV) or CTRL-V (mode noCTRL-V). *Refer mode().* All the modes handled are listed at Mode Names and All of Vim's modes demonstrated shows them in action.

9.6. Statusline commands

These commands build the statusline, with a series of appending commands, using ternary expressions for conditional components. They start with the mode names and their applicable highlight groups. The default and user-determined components are then appended.

To illustrate, the following line of code adds a 'b' and the buffer number to the statusline, provided the variable g:tene_buffer_num equals 1:

```
set statusline+=%{g:tene_buffer_num==1?'b'..bufnr('%')..'\ ':''}
```

Being a ternary operator-driven expression, the false condition needs to be specified, so when g:tene_buffer_num does not equal 1 the addition to the statusline is '', i.e., nothing.

This example also illustrates another design decision: in most cases, using Vim's builtin functions — here bufnr() — are used versus the shorthand statusline item (which in this case is %n). The following are synonymous:

```
set statusline+=%{g:tene_buffer_num==1?'b'..bufnr('%')..'\ ':''}
set statusline+=%{%g:tene_buffer_num==1?'b%n\ ':''%}
```

Reasons for preferring the more verbose bufnr() are:

1. It is easier to see what's being done in the code, i.e., bufnr() versus %n, which requires you to look up the help, whereas bufnr() you can infer means "buffer number", and

2. %{expression} rather than %{%expression%} is more readable. It is easy to either omit or include % signs.

This a simple example, though conceptually all of the ternary expressions are like this. Some are nested and/or have more than one setting or variable in scope. And some are quite long.

9.7. Enabling <Plug> commands for toggling variables

There are several variables that may be toggled — *see* Variables that may be toggled. They may be:

- left as their defaults, and/or
- be set in your ~/.vimrc, and/or
- toggled interactively.

In terms of toggling interactively, the `map <Plug>` commands do that. And, predictably, they execute ternary expressions! They also follow the approach outlined in using-<Plug>, because it is possible that you may want to map your own key(s) to a mapping(s).

An example:

```
map <Plug>TeneB <Cmd>execute "let g:tene_buffer_num = (g:tene_buffer_num ==
1) ? 0 : 1"<CR>
```

All the mappings follow this structure, using the example above to explain:

map	Maps the key sequence (in Normal, Visual, Select, and Operator-pending modes) — *refer* map-table
	ℹ️ Often :map should be avoided, with :noremap usually being advisable but, because this is a <Plug> mapping, that's unnecessary. *Refer also* using-<Plug>.
<Plug>	Avoids typed key mappings, and is available outside the script
TeneB	This is a unique name (the script name + char(s)) — *refer* using-<Plug>
<Cmd>	<Cmd> starts a "command mapping", without changing modes in Visual and Operator-pending modes
execute	Executes the string that follows it as an Ex command
"let..."<CR>	The ternary expression toggling the variable

See Binary variables for toggling features for the list of variables that may be toggled.

9.8. Leader mappings

To use the <Plug> mappings, explained above, each <Plug> mapping is itself mapped to a <Leader> key sequence. There are two exceptions though:

1. Only if you have not already mapped the applicable <Plug> mapping. That is, a further mapping won't be added. That is because it would be rare to want multiple mappings doing the same thing.

2. Only if the <Leader> mapping, which *would* be created, doesn't exist already. That is unlikely though. All the default <Leader> mappings are two keys, i.e., <Leader>t*character*). Nonetheless, overwriting any of your mappings should be avoided! (That is consistent with aim #4, respect users' settings.)

An example follows, mapping <Leader>tb to <Plug>TeneB, which toggles the display of buffer numbers.

```
execute (!hasmapto('<Plug>TeneB') && maparg('<Leader>tb', '') == '') ? ':map
<Leader>tb <Plug>TeneB' : ''
```

execute	Executes the following Ex command.
!hasmapto('<Plug>TeneB')	Tests for whether <Plug>TeneB has been mapped already.
maparg('<Leader>tb', '')	Uses maparg to test for whether <Leader>tb has been mapped already.[22]
:map<Leader>tb <Plug>TeneB	Maps (in Normal, Visual, Select, and Operator-pending modes) <Leader>tb to <Plug>TeneB. (<Plug>TeneB toggles g:tene_buffer_num — *see* Default <Leader> mappings)
: ''	This is the "do nothing" part of the ternary expression.

9.8.1. Default <Leader> mappings

ℹ️   For an animated .gif showing this in action, *see* Illustrating toggling variables

<Leader>	<Plug>	Display Toggles "on"	"off"
t%	Tene%	Percent (at the cursor) through the buffer	*nothing*
tb	TeneB	b{buffer number} after mode indicator	*nothing*
tc	TeneC	col() (and -{num} if %V is different)	*nothing*
tf	TeneF	File name only	Relative or full path plus file name
tg	TeneG	Unicode glyphs for line number, etc.	ASCII chars
th	TeneH	Highlight group under the cursor	*nothing*
tk	TeneK	Indicate "b:keymap_name" in mode label	*nothing*
tl	TeneL	Line number (of the cursor)	*nothing*
tm	TeneM	NORMAL, INSERT, VISUAL LINE, EX	n, i, V, ce
tp	TeneP	Full file path of the buffer	Relative file path
ts	TeneS	mode(1) and state() codes	*nothing*
tu	TeneU	U+nnnnn of character(s) at the cursor	*nothing*
tv	TeneV	Virtual column number, virtcol()	*nothing*
tz	TeneZ	Total number of lines in the buffer	*nothing*

Configuring these to use different <Leader> *characters* to the defaults is explained in Leader keys.

And that's it, THE END.

THE END?

Well, other than the modeline, with foldmethod=marker, which makes folds apply automatically according to the included markers, {{{ ... }}}. And, to wrap it up, literally, there is 'nowrap'. Although long lines have drawbacks, there are benefits too, e.g., using i_CTRL-Y and i_CTRL-E when editing lines with similar content.

10. Tene

Why call the plugin "tene"? Tene, in Māori, means "impromptu", "improvised", or "spontaneous". I had no intention of writing a statusline plugin: it came about as I looked for a way to have a ternary expressions 'statusline' in my ~/.vimrc and it escalated from there. Impromptu? Sure. I wanted to get away from bloated statusline plugins, which, although sometimes feature-heavy, can be slow, can be hard to follow what's going on (with heaps of user-defined functions spread across lots of files), and try to do lots of things to accommodate other plugins.

Avoiding "line" directly in the name was incidental because, although it's useful for identification as a statusline plugin, I made the deliberate decision to use only ternary expressions (and also no user-defined functions), "tene" works in that regard too: it's a **te**rnary statusli**ne**.

11. Licence

BSD 3-Clause License. Copyright © 2023 Peter Kenny

Endnotes

[1] That is, Windows versions between v8.2.3557 and v8.2.5171, and Unix versions between v8.2.3555 and v8.2.5172.

[2] I tested Windows v8.2.5171 and built and tested on WSL Debian 11, a laptop with Debian 11, and a Raspberry Pi Zero (Debian 10.13) using v8.2.5172.

[3] *Refer*, for example, GitHub for the places ModeChanged has been patched to learn the reasons for this.

[4] Version 8.2.3434: Windows and Unix.

[5] *Refer* Neovim builtin functions. It is not hard to remove some code (and consequently, functionality) that relies on state(), add `let` to variables, etc., and that would make a *not-as-featured* Neovim version. In fact, I started doing it just to validate it is feasible; broadly, it is. There also appears to be a redrawing issue with Neovim in that updating the mode indicator appears to be delayed, but there is probably a solution for that (perhaps another autocommand(s)). However, because I do not use Neovim, and it would sufficiently diverge to being a different plugin, someone else can look at doing that.

[6] Incidentally, Vim's help considers expressions using ternary operators the "least … significant" of Vim's expressions. Not in this plugin! *Refer* expression-syntax.

[7] There is always ga or :ascii to get that information in this instance, but that requires keystrokes, especially when not in Normal mode.

[8] The Unicode codes only are indicated here because fonts that display them cannot be used, as far as I'm aware, on Github.

[9] There was a conundrum determining this. Other "cut off" points could logically have been chosen. One such point could have been patch 3965, 2022-01-02, "Vim9: no easy way to check if Vim9 script is supported", when `has('vim9script')` first returns 1. Another possibility was, if :def functions had been utilised, patch 4615, 2022-03-24, when "mapping with escaped bar does not work in :def function" was fixed. In the end I chose the ModeChanged autocommand event date (patch 8.2.3434, 2021-09-13, "function prototype for trigger_modechanged() is incomplete") because that was the critical addition enabling detection of mode changes, though it was a close call with patch 3555, because not all mode changes were detectable until then (2021-10-23). And since Vim9 script was sufficiently stable by that time, choosing to use it was an arbitrary, personal, choice.

[10] Clearly the nested ternary expressions are complex. They even may be viewed as an abomination by some. Nonetheless, *conceptually*, the end result is simple.

[11] That's not hating on you if you do use tabs like buffers — it's your choice — you do you. ☺

[12] This relies on the ModeChanged autocommand event enabled with version 8.2 patches 3434 (2021-09-13) and 3555 (2021-10-23). *See* Autocommand group. Also *refer* Pull request 8856.

[13] The expected 10x to 100x speed increase delivered by Vim9 script is not critical to this plugin. That's because once the statusline has been set, and configuration options are applied, vim-tene's job is done. It is not a script that has functions called repeatedly nor does it perform any complex actions, e.g., substituting a complex pattern across a large buffer(s), which, from personal experience there is a huge speed performance gain versus vimscript.

[14] On a mid-high spec Windows 10 desktop (Intel Core i7-6700, 32GB RAM) it was always 2ms (Vim and gvim using --startuptime). That doubled to 4ms-5ms on a lower spec Debian 11 laptop (AMD A8 7140, 8GB RAM), so still imperceptible. On a Raspberry Pi Zero that jumped to 26ms, but that was expected because all sourced .vim files took around 10x-15x longer on the Pi — that's hardly suprising for an ARM1176 and only 500MB RAM! (As an aside, Vim on the Pi seems picky, even defective at times: I could not get any packadd to

work on it — the only way I could get tene.vim (or any plugin or .vim file) to load was to put it in the `/usr/local/share/vim/vim82/plugin` directory. That was despite having identical built versions (2.8 with patch 5172), identical `~/.vimrc` files, and identical `pack` directory structures.)

[15] The virtual column, virtcol('.') differs from either %v or %V. Should %v show the same thing? It *almost* does, except for where a character is one that consumes more than a column (i.e., it displays the same for multi-byte characters, but differently where the character is considered one, like <Tab>, but which can consume more than one character space. It is very close, because it is the same after the <Tab> but is different when the cursor is sitting at the start of the <Tab>.)

[16] The way modes change is not always direct. For example, if going from n to no to nov by keying dv, the mode transition is: n, (d) no, (v) n, nov. It's not clear why that's the case, i.e., the "extra" **n** between no and nov, but it can be shown/proven by entering the following command after starting Vim: `:autocmd ModeChanged *` `call popup_menu(mode(1),#{time: 2000})`, which will generate a two second popup with every mode change.

[17] This could possibly be restricted to changing to c* and no* modes because it seems that changing to other modes is detected and applied anyway. But, there appears to be little downside in redrawing the statusline, aside from the overhead in doing so. I found that imperceptable on my desktop and laptop, so decided the "insurance", leaving the command as [`^t`], was acceptable. Even on the Raspberry Pi Zero, surely one of the lowest powered devices you are likely to use today, the performance was imperceptibly different. However, a very noticable delay, e.g., in updating NORMAL when keying `<Esc>` in Insert mode was there regardless of the autocommand. If you really do want to only apply the autocommand when entering Ex, Vim Ex, or any of the Operator-pending modes (no, plus nov, noV and noCTRL-V), replace the single autocommand with: `autocmd ModeChanged *:c* redrawstatus` and `autocmd ModeChanged *:no* redrawstatus`.

[18] An example is where CTRL-O CTRL-W w is used to go to the terminal from a buffer in Insert mode. If redrawstatus is executed, 'showmode' displays -- **(insert)** -- and that will persist, even if you return to the window that is in Insert mode.)

[19] This is a bit of a "hack" insofar as the default highlight groups leveraged are unrelated to the mode indicators, but results of testing were good, i.e., the indicators of modes seemed to display well with either a "light" or "dark" background, using Vim 9's collection of colorschemes.

[20] The details and a discussion of this, including why, in some cases it *may* be wanted, can be found at https://github.com/vim/vim/issues/12072.

[21] This has been tested all the way back to Win32 console version 7.2, 2008-08-09. Almost 15 years! That's far enough. Also note, if below version 704 with patch 1485, `source` rather than `packadd` is needed.

[22] Incidentally, initially this was coded as `maparg(g:mapleader .. 'tb')`, but on at least one O/S (Raspberry Pi, Debian 10.13) it produced an error, because `g:mapleader` was not recognised as a global variable. On every other O/S it was so, because `'<Leader>tb'`, `'<Space>tb'` and `g:mapleader .. 'tb'` all work (when <Space> is the Leader key), that is what the code was changed to. And, if your <Leader> key is <Space>, `echo g:mapleader` (on all the other O/Ss) outputs `''`, so it appears like it is nothing.