Transient User and Developer Manual

for version 0.11.0

Jonas Bernoulli

Table of Contents

1 Introduction

Transient is the library used to implement the keyboard-driven *menus* in Magit. It is distributed as a separate package, so that it can be used to implement similar menus in other packages.

This manual can be bit hard to digest when getting started. A useful resource to get over that hurdle is Psionic K's interactive tutorial, available at `https://github.com/positron-solutions/transient-showcase`.

Some things that Transient can do

- Display current state of arguments
- Display and manage lifecycle of modal bindings
- Contextual user interface
- Flow control for wizard-like composition of interactive forms
- History & persistence
- Rendering arguments for controlling CLI programs

Complexity in CLI programs

Complexity tends to grow with time. How do you manage the complexity of commands? Consider the humble shell command '`ls`'. It now has over *fifty* command line options. Some of these are boolean flags ('`ls -l`'). Some take arguments ('`ls --sort=s`'). Some have no effect unless paired with other flags ('`ls -lh`'). Some are mutually exclusive. Some shell commands even have so many options that they introduce *subcommands* ('`git branch`', '`git commit`'), each with their own rich set of options ('`git branch -f`').

Using Transient for composing interactive commands

What about Emacs commands used interactively? How do these handle options? One solution is to make many versions of the same command, so you don't need to! Consider: '`delete-other-windows`' vs. '`delete-other-windows-vertically`' (among many similar examples).

Some Emacs commands will simply prompt you for the next "argument" ('`M-x switch-to-buffer`'). Another common solution is to use prefix arguments which usually start with '`C-u`'. Sometimes these are sensibly numerical in nature ('`C-u 4 M-x forward-paragraph`' to move forward 4 paragraphs). But sometimes they function instead as boolean "switches" ('`C-u C-SPACE`' to jump to the last mark instead of just setting it, '`C-u C-u C-SPACE`' to unconditionally set the mark). Since there aren't many standards for the use of prefix options, you have to read the command's documentation to find out what the possibilities are.

But when an Emacs command grows to have a truly large set of options and arguments, with dependencies between them, lots of option values, etc., these simple approaches just don't scale. Transient is designed to solve this issue. Think of it as the humble prefix argument '`C-u`', *raised to the power of 10*. Like '`C-u`', it is key driven. Like the shell, it supports boolean "flag" options, options that take arguments, and even "sub-commands", with their

own options. But instead of searching through a man page or command documentation, well-designed transients *guide* their users to the relevant set of options (and even their possible values!) directly, taking into account any important pre-existing Emacs settings. And while for shell commands like '`ls`', there is only one way to "execute" (hit '`Return`'!), transients can "execute" using multiple different keys tied to one of many self-documenting *actions* (imagine having 5 different colored return keys on your keyboard!). Transients make navigating and setting large, complex groups of command options and arguments easy. Fun even. Once you've tried it, it's hard to go back to the '`C-u what can I do here again?`' way.

2 Usage

2.1 Invoking Transients

A transient prefix command is invoked like any other command by pressing the key that is bound to that command. The main difference to other commands is that a transient prefix command activates a transient keymap, which temporarily binds the transient's infix and suffix commands, and that those bindings are shown in menu buffer, which is displayed in a new window, until the menu is exited. Bindings from other keymaps may, or may not, be disabled while the transient state is in effect.

There are two kinds of commands that are available after invoking a transient prefix command; infix and suffix commands. Infix commands set some value (which is then shown in the menu buffer), without leaving the transient. Suffix commands, on the other hand, usually quit the transient and they may use the values set by the infix commands, i.e., the infix **arguments**.

Instead of setting arguments to be used by a suffix command, infix commands may also set some value by side-effect, e.g., by setting the value of some variable.

2.2 Aborting and Resuming Transients

To quit the transient without invoking a suffix command press `C-g`.

Key bindings in transient keymaps may be longer than a single event. After pressing a valid prefix key, all commands whose bindings do not begin with that prefix key are temporarily unavailable and grayed out. To abort the prefix key press `C-g` (which in this case only quits the prefix key, but not the complete transient).

A transient prefix command can be bound as a suffix of another transient. Invoking such a suffix replaces the current transient state with a new transient state, i.e., the available bindings change and the information displayed in the menu buffer is updated accordingly. Pressing `C-g` while a nested transient is active only quits the innermost transient, causing a return to the previous transient.

`C-q` or `C-z` on the other hand always exits all transients. If you use the latter, then you can later resume the stack of transients using `M-x transient-resume`.

`C-g` (`transient-quit-seq`)
`C-g` (`transient-quit-one`)
: This key quits the currently active incomplete key sequence, if any, or else the current transient. When quitting the current transient, it returns to the previous transient, if any.

Transient's predecessor bound `q` instead of `C-g` to the quit command. To learn how to get that binding back see `transient-bind-q-to-quit`'s documentation string.

`C-q` (`transient-quit-all`)
: This command quits the currently active incomplete key sequence, if any, and all transients, including the active transient and all suspended transients, if any.

`C-z` (`transient-suspend`)
: Like `transient-quit-all`, this command quits an incomplete key sequence, if any, and all transients. Additionally, it saves the stack of transients so that it

can easily be resumed (which is particularly useful if you quickly need to do "something else" and the stack is deeper than a single transient, and/or you have already changed the values of some infix arguments).

Note that only a single stack of transients can be saved at a time. If another stack is already saved, then saving a new stack discards the previous stack.

M-x transient-resume
: This command resumes the previously suspended stack of transients, if any.

2.3 Common Suffix Commands

A few shared suffix commands are available in all transients. These suffix commands are not shown permanently in every menu by default. Most of these commands share a common prefix key and pressing that key causes the common commands to be temporarily shown in the active menu.

`transient-show-common-commands` [User Option]
: This option controls whether shared suffix commands are permanently shown alongside the menu-specific infix and suffix commands. By default, the shared commands are not permanently shown to avoid wasting precious space and overwhelming the user with too many choices.

  If you prefer to always see these commands, then set this option to a non-`nil` value. Alternatively the value can be toggled for the current Emacs session only, using `transient-toggle-common`, described below.

`transient-common-command-prefix` [User Option]
: This option specifies the prefix key used in all transient menus to invoke most of the shared commands, which are available in all transient menus. By default these bindings are only shown after pressing that prefix key and before following that up with a valid key binding (but see the previous option).

  For historic reasons *C-x* is used by default, but users are encouraged to pick another key, preferably one that is not commonly used in Emacs but is still convenient to them.

  Usually, while a transient menu is active, the user cannot invoke commands that are not bound in the menu itself. For those menus it does not matter, if *C-x* or another commonly used prefix key is used for common menu commands. However, certain other, newer menus do not suppress key bindings established outside the menu itself, and in those cases a binding for a common menu command could shadow an external binding. For example, *C-x C-s* could not be used to invoke `save-buffer`, if that binding is shadowed by the menu binding for `transient-save`.

  Which key is most suitable depends on the user's preferences, but good choices may include function keys and *C-z* (for many keyboard layouts *z* is right next to *x*, and invoking `suspend-frame`, while a transient menu is active, would not be a good idea anyway).

C-x t (`transient-toggle-common`)
: This command toggles whether the generic commands, that are common to all transients, are permanently displayed or only after typing the incomplete prefix key sequence. This only affects the current Emacs session.

The other common commands are described in either the previous or in one of the following sections.

2.4 Saving Values

After setting the infix arguments in a transient, the user can save those arguments for future invocations.

Most transients will start out with the saved arguments when they are invoked. There are a few exceptions, though. Some transients are designed so that the value that they use is stored externally as the buffer-local value of some variable. Invoking such a transient again uses the buffer-local value.[1]

If the user does not save the value and just exits using a regular suffix command, then the value is merely saved to the transient's history. That value won't be used when the transient is next invoked, but it is easily accessible (see Section 2.5 [Using History], page 5).

Option `transient-common-command-prefix` controls the prefix key used in the following bindings. For simplicity's sake the default, `C-x`, is shown below.

`C-x s` (`transient-set`)
: This command saves the value of the active transient for this Emacs session.

`C-x C-s` (`transient-save`)
: This command saves the value of the active transient persistently across Emacs sessions.

`C-x C-k` (`transient-reset`)
: This command clears the set and saved values of the active transient.

`transient-values-file` [User Option]
: This option names the file that is used to persist the values of transients between Emacs sessions.

2.5 Using History

Every time the user invokes a suffix command the transient's current value is saved to its history. These values can be cycled through, the same way one can cycle through the history of commands that read user-input in the minibuffer.

Option `transient-common-command-prefix` controls the prefix key used in the following bindings. For simplicity's sake the default, `C-x`, is shown below.

`C-M-p` (`transient-history-prev`)
`C-x p` This command switches to the previous value used for the active transient.

`C-M-n` (`transient-history-next`)
`C-x n` This command switches to the next value used for the active transient.

In addition to the transient-wide history, infixes can have their own history. When an infix reads user-input using the minibuffer, the user can use the regular minibuffer

[1] `magit-diff` and `magit-log` are two prominent examples, and their handling of buffer-local values is actually a bit more complicated than outlined above and even customizable.

history commands to cycle through previously used values. Usually the same keys as those mentioned above are bound to those commands.

Authors of transients should arrange for different infix commands that read the same kind of value to also use the same history key (see Section 5.8 [Suffix Slots], page 38).

Both kinds of history are saved to a file when Emacs is exited.

`transient-save-history` [User Option]
: This option controls whether the history of transient commands is saved when exiting Emacs.

`transient-history-file` [User Option]
: This option names the file that is used to persist the history of transients and their infixes between Emacs sessions.

`transient-history-limit` [User Option]
: This option controls how many history elements are kept at the time the history is saved in `transient-history-file`.

2.6 Getting Help for Suffix Commands

Transients can have many suffixes and infixes that the user might not be familiar with. To make it trivial to get help for these, Transient provides access to the documentation directly from the active transient.

`C-h` (`transient-help`)
: This command enters help mode. When help mode is active, typing a key shows information about the suffix command that the key normally is bound to (instead of invoking it). Pressing `C-h` a second time shows information about the *prefix* command.

  After typing a key, the stack of transient states is suspended and information about the suffix command is shown instead. Typing `q` in the help buffer buries that buffer and resumes the transient state.

What sort of documentation is shown depends on how the transient was defined. For infix commands that represent command-line arguments this ideally shows the appropriate manpage. `transient-help` then tries to jump to the correct location within that. Info manuals are also supported. The fallback is to show the command's documentation string, for non-infix suffixes this is usually appropriate.

2.7 Enabling and Disabling Suffixes

The user base of a package that uses transients can be very diverse. This is certainly the case for Magit; some users have been using it and Git for a decade, while others are just getting started now.

For that reason a mechanism is needed that authors can use to classify a transient's infixes and suffixes along the essentials. . . everything spectrum. We use the term *levels* to describe that mechanism.

Each suffix command is placed on a level and each transient has a level (called *transient-level*), which controls which suffix commands are available. Integers between 1 and 7 (inclusive) are valid levels. For suffixes, 0 is also valid; it means that the suffix is not displayed at any level.

The levels of individual transients and/or their individual suffixes can be changed interactively, by invoking the menu and entering its "edit" mode using the command `transient-set-level`, as described below.

The default level for both transients and their suffixes is 4. The `transient-default-level` option only controls the default for transients. The default suffix level is always 4. The authors of transients should place certain suffixes on a higher level, if they expect that it won't be of use to most users, and they should place very important suffixes on a lower level, so that they remain available even if the user lowers the transient level.

`transient-default-level` [User Option]

This option controls which suffix levels are made available by default. It sets the transient-level for transients for which the user has not set that individually.

`transient-levels-file` [User Option]

This option names the file that is used to persist the levels of transients and their suffixes between Emacs sessions.

Option `transient-common-command-prefix` controls the prefix key used in the following bindings. For simplicity's sake the default, `C-x`, is shown below.

`C-x l` (`transient-set-level`)

This command enters edit mode. When edit mode is active, then all infixes and suffixes that are currently usable are displayed along with their levels. The colors of the levels indicate whether they are enabled or not. The level of the transient is also displayed along with some usage information.

In edit mode, pressing the key that would usually invoke a certain suffix instead prompts the user for the level that suffix should be placed on.

Help mode is available in edit mode.

To change the transient level press `C-x l` again.

To exit edit mode press `C-g`.

Note that edit mode does not display any suffixes that are not currently usable. `magit-rebase`, for example, shows different suffixes depending on whether a rebase is already in progress or not. The predicates also apply in edit mode.

Therefore, to control which suffixes are available given a certain state, you have to make sure that that state is currently active.

`C-x a` (`transient-toggle-level-limit`)

This command toggle whether suffixes that are on levels higher than the level specified by `transient-default-level` are temporarily available anyway.

`transient-set-default-level` *suffix level* [Function]

This function sets the default level of the suffix COMMAND to LEVEL.

If a suffix command appears in multiple menus, it may make sense to consistently change its level in all those menus at once. For example, the `--gpg-sign` argument

(which is implemented using the command `magit:--gpg-sign`), is bound in all of Magit's menu which create commits. Users who sometimes sign their commits would want that argument to be available in all of these menus, while for users who never sign it is just unnecessary noise in any menus.

To always make `--gpg-sign` available, use:

```
(transient-set-default-level 'magit:--gpg-sign 1)
```

To never make `--gpg-sign` available, use:

```
(transient-set-default-level 'magit:--gpg-sign 0)
```

This sets the level in the suffix prototype object for this command. Commands only have a suffix prototype if they were defined using one of `transient-define-argument`, `transient-define-infix` and `transient-define-suffix`. For all other commands this would signal an error. (This is one of the reasons why package authors should use one of these functions to define shared suffix commands, and especially shared arguments.)

If the user changes the level of a suffix in a particular menu, using `C-x l` as shown above, then that obviously shadows the default.

It is also possible to set the level of a suffix binding in a particular menu, either when defining the menu using `transient-define-prefix,` or later using `transient-insert-suffix`. If such bindings specify a level, then that also overrides the default. (Per-suffix default levels is a new feature, so you might encounter this quite often.)

2.8 Other Commands

When invoking a transient in a small frame, the transient window may not show the complete buffer, making it necessary to scroll, using the following commands. These commands are never shown in the transient window, and the key bindings are the same as for `scroll-up-command` and `scroll-down-command` in other buffers.

`transient-scroll-up` *arg* [Command]
: This command scrolls text of transient's menu window upward *ARG* lines. If *ARG* is `nil`, then it scrolls near full screen. This is a wrapper around `scroll-up-command` (which see).

`transient-scroll-down` *arg* [Command]
: This command scrolls text of transient's menu window down *ARG* lines. If *ARG* is `nil`, then it scrolls near full screen. This is a wrapper around `scroll-down-command` (which see).

The following commands are not available by default. If you would like to use them for all menus, bind them in `transient-map`.

`transient-copy-menu-text` [Command]
: This command copies the contents of the menu buffer to the kill ring.

`transient-toggle-docstrings` [Command]
: This command toggle between showing suffix descriptions in the menu (as usual) or showing the first lines of the respective docstrings in their place. For commands that

do not have a docstring, always display the suffix description. Because there likely isn't enough room to display multiple docstrings side-by-side, a single column is used when displaying docstrings.

2.9 Configuration

More options are described in Section 2.3 [Common Suffix Commands], page 4, in Section 2.4 [Saving Values], page 5, in Section 2.5 [Using History], page 5, and in Section 2.7 [Enabling and Disabling Suffixes], page 6.

Essential Options

Two more essential options are documented in Section 2.3 [Common Suffix Commands], page 4.

`transient-show-popup` [User Option]

This option controls whether and when transient's menu buffer is shown.

- If `t` (the default), then the buffer is shown as soon as a transient prefix command is invoked.
- If `nil`, then the buffer is not shown unless the user explicitly requests it, by pressing an incomplete prefix key sequence.
- If a number, then the a brief one-line summary is shown instead of the menu buffer. If zero or negative, then not even that summary is shown; only the pressed key itself is shown.

  The buffer is shown once the user explicitly requests it by pressing an incomplete prefix key sequence. Unless this is zero, the menu is shown after that many seconds of inactivity (using the absolute value).

`transient-show-during-minibuffer-read` [User Option]

This option controls whether the transient menu continues to be displayed while the minibuffer is used to read user input.

This is only relevant to commands that do not close the menu, such as commands that set infix arguments. If a command exits the menu, and uses the minibuffer, then the menu is always closed before the minibuffer is entered, irrespective of the value of this option.

When `nil` (the default), hide the menu while the minibuffer is in use. When `t`, keep showing the menu, but allow for the menu window to be resized, to ensure that completion candidates can be displayed.

When `fixed`, keep showing the menu and prevent it from being resized, which may make it impossible to display the completion candidates. If that ever happens for you, consider using `t` or an integer, as described below.

If the value is `fixed` and the menu window uses the full height of its frame, then the former is ignored and resizing is allowed anyway. This is necessary because individual menus may use unusual display actions different from what `transient-display-buffer-action` specifies (likely to display that menu in a side-window).

When using a third-party mode, which automatically resizes windows (e.g., by calling `balance-windows` on `post-command-hook`), then `fixed` (or `nil`) is likely a better choice than `t`.

The value can also be an integer, in which case the behavior depends on whether at least that many lines are left to display windows other than the menu window. If that is the case, display the menu and preserve the size of that window. Otherwise, allow resizing the menu window if the number is positive, or hide the menu if it is negative.

`transient-read-with-initial-input` [User Option]
This option controls whether the last history element is used as the initial minibuffer input when reading the value of an infix argument from the user. If `nil`, there is no initial input and the first element has to be accessed the same way as the older elements.

`transient-enable-popup-navigation` [User Option]
This option controls whether navigation commands are enabled in transient's menu buffer. If the value is `verbose` (the default), brief documentation about the command under point is additionally show in the echo area.

While a transient is active the menu buffer is not the current buffer, making it necessary to use dedicated commands to act on that buffer itself. If this option is non-`nil`, then the following features are available:

- `UP` moves the cursor to the previous suffix.
- `DOWN` moves the cursor to the next suffix.
- *`M`*`-RET` invokes the suffix the cursor is on.
- *`mouse-1`* invokes the clicked on suffix.
- *`C-s`* and *`C-r`* start isearch in the menu buffer.

By default *`M`*`-RET` is bound to `transient-push-button`, instead of `RET`, because if a transient allows the invocation of non-suffixes, then it is likely, that you would want `RET` to do what it would do if no transient were active."

`transient-display-buffer-action` [User Option]
This option specifies the action used to display the transient's menu buffer. The menu buffer is displayed in a window using `(display-buffer` *`BUFFER`* `transient-display-buffer-action)`.

The value of this option has the form `(`*`FUNCTION`*` . `*`ALIST`*`)`, where *FUNCTION* is a function or a list of functions. Each such function should accept two arguments: a buffer to display and an alist of the same form as *ALIST*. See Section "Choosing Window" in `elisp`, for details.

The default is:

```
(display-buffer-in-side-window
 (side . bottom)
 (dedicated . t)
 (inhibit-same-window . t))
```

This displays the window at the bottom of the selected frame. For alternatives see Section "Buffer Display Action Functions" in `elisp`, and Section "Buffer Display Action Alists" in `elisp`.

When you switch to a different ACTION, you should keep the ALIST entries for `dedicated` and `inhibit-same-window` in most cases. Do not drop them because you are unsure whether they are needed; if you are unsure, then keep them.

Note that the buffer that was current before the transient buffer is shown should remain the current buffer. Many suffix commands act on the thing at point, if appropriate, and if the transient buffer became the current buffer, then that would change what is at point. To that effect `inhibit-same-window` ensures that the selected window is not used to show the transient buffer.

It may be possible to display the window in another frame, but whether that works in practice depends on the window-manager. If the window manager selects the new window (Emacs frame), then that unfortunately changes which buffer is current.

If you change the value of this option, then you might also want to change the value of `transient-mode-line-format`.

This user option may be overridden if `:display-action` is passed when creating a new prefix with `transient-define-prefix`.

Accessibility Options

`transient-force-single-column` [User Option]

This option controls whether the use of a single column to display suffixes is enforced. This might be useful for users with low vision who use large text and might otherwise have to scroll in two dimensions.

Auxiliary Options

`transient-mode-line-format` [User Option]

This option controls whether transient's menu buffer has a mode-line, separator line, or neither.

If `nil`, then the buffer has no mode-line. If the buffer is not displayed right above the echo area, then this probably is not a good value.

If `line` (the default) or a natural number, then the buffer has no mode-line, but a line is drawn in its place. If a number is used, that specifies the thickness of the line. On termcap frames we cannot draw lines, so there `line` and numbers are synonyms for `nil`.

The color of the line is used to indicate if non-suffixes are allowed and whether they exit the transient. The foreground color of `transient-key-noop` (if non-suffixes are disallowed), `transient-key-stay` (if allowed and transient stays active), or `transient-key-exit` (if allowed and they exit the transient) is used to draw the line.

This user option may be overridden if `:mode-line-format` is passed when creating a new prefix with `transient-define-prefix`.

Otherwise this can be any mode-line format. See Section "Mode Line Format" in `elisp`, for details.

`transient-semantic-coloring` [User Option]

This option controls whether colors are used to indicate the transient behavior of commands.

If non-`nil`, then the key binding of each suffix is colorized to indicate whether it exits the transient state or not. The color of the prefix is indicated using the line that is drawn when the value of `transient-mode-line-format` is `line`.

`transient-highlight-mismatched-keys` [User Option]

This option controls whether key bindings of infix commands that do not match the respective command-line argument should be highlighted. For other infix commands this option has no effect.

This is mostly indended for autors of transient menus and disabled by default.

When this option is non-`nil`, the key binding for an infix argument is highlighted when only a long argument (e.g., `--verbose`) is specified but no shorthand (e.g., `-v`). In the rare case that a shorthand is specified but the key binding does not match, then it is highlighted differently.

Highlighting mismatched key bindings is useful when learning the arguments of the underlying command-line tool; you wouldn't want to learn any short-hands that do not actually exist.

The highlighting is done using one of the faces `transient-mismatched-key` and `transient-nonstandard-key`.

`transient-substitute-key-function` [User Option]

This function is used to modify key bindings. If the value of this option is `nil` (the default), then no substitution is performed.

This function is called with one argument, the prefix object, and must return a key binding description, either the existing key description it finds in the `key` slot, or the key description that replaces the prefix key. It could be used to make other substitutions, but that is discouraged.

For example, `=` is hard to reach using my custom keyboard layout, so I substitute `(` for that, which is easy to reach using a layout optimized for lisp.

```
(setq transient-substitute-key-function
      (lambda (obj)
        (let ((key (oref obj key)))
          (if (string-match "\\`\\(=\\)[a-zA-Z]" key)
              (replace-match "(" t t key 1)
            key))))
```

`transient-align-variable-pitch` [User Option]

This option controls whether columns are aligned pixel-wise in the menu buffer.

If this is non-`nil`, then columns are aligned pixel-wise to support variable-pitch fonts. Keys are not aligned, so you should use a fixed-pitch font for the `transient-key` face. Other key faces inherit from that face unless a theme is used that breaks that relationship.

This option is intended for users who use a variable-pitch font for the `default` face.

`transient-force-fixed-pitch` [User Option]

This option controls whether to force the use of a monospaced font in menu buffer. Even if you use a proportional font for the `default` face, you might still want to use a monospaced font in the menu buffer. Setting this option to `t` causes `default` to be remapped to `fixed-pitch` in that buffer.

Developer Options

These options are mainly intended for developers.

`transient-detect-key-conflicts` [User Option]

This option controls whether key binding conflicts should be detected at the time the transient is invoked. If so, this results in an error, which prevents the transient from being used. Because of that, conflicts are ignored by default.

Conflicts cannot be determined earlier, i.e., when the transient is being defined and when new suffixes are being added, because at that time there can be false-positives. It is actually valid for multiple suffixes to share a common key binding, provided the predicates of those suffixes prevent that more than one of them is enabled at a time.

`transient-error-on-insert-failure` [User Option]

This option controls whether to signal an error when `transient-insert-suffix` or `transient-append-suffix` failed to insert a suffix into an existing prefix. By default a warning is shown instead.

`transient-highlight-higher-levels` [User Option]

This option controls whether suffixes that would not be available by default are highlighted.

When non-`nil` then the descriptions of suffixes are highlighted if their level is above 4, the default of `transient-default-level`. Assuming you have set that variable to 7, this highlights all suffixes that won't be available to users without them making the same customization.

Hook Variables

`transient-exit-hook` [Variable]

This hook is run after a transient menu is exited, even if another transient menu becomes active at the same time.

`transient-post-exit-hook` [Variable]

This hook is run after a transient menu is exited, provided no other transient menu becomes active at the same time.

`transient-setup-buffer-hook` [Variable]

This hook is run when the transient buffer is being setup. That buffer is current and empty when this hook is runs.

3 Modifying Existing Transients

To an extent, transients can be customized interactively, see Section 2.7 [Enabling and Disabling Suffixes], page 6. This section explains how existing transients can be further modified non-interactively. Let's begin with an example:

```
(transient-append-suffix 'magit-patch-apply "-3"
  '("-R" "Apply in reverse" "--reverse"))
```

This inserts a new infix argument to toggle the `--reverse` argument after the infix argument that is bound to `-3` in `magit-patch-apply`.

The following functions share a few arguments:

- *PREFIX* is a transient prefix command, a symbol.

  PREFIX may also by a symbol identifying a separately defined group, which can be included in multiple prefixes. See TODO.

- *SUFFIX* is a transient infix or suffix specification in the same form as expected by `transient-define-prefix`. Note that an infix is a special kind of suffix. Depending on context "suffixes" means "suffixes (including infixes)" or "non-infix suffixes". Here it means the former. See Section 4.3.2 [Suffix Specifications], page 21.

  SUFFIX may also be a group in the same form as expected by `transient-define-prefix`. See Section 4.3.1 [Group Specifications], page 18.

- *LOC* is a key description (a string as returned by `key-description` and understood by `kbd`), a command, a symbol identifying an included group, or a vector specifying coordinates. For example, `[1 0 -1]` identifies the last suffix (`-1`) of the first subgroup (`0`) of the second group (`1`).

  If *LOC* is a vector, then it can be used to identify a group, not just an individual suffix command. The last element in a vector may also be a symbol or key, in which case the preceding elements must match a group and the last element is looked up within that group.

  The function `transient-get-suffix` can be useful to determine whether a certain coordinate vector identifies the suffix or group that you expect it to identify. In hairy cases it may be necessary to look at the internal layout representation, which you can access using the function `transient--get-layout`.

These functions operate on the information stored in the `transient--layout` property of the *PREFIX* symbol. Elements in that tree are not objects but have the form `(`*`CLASS PLIST`*`) for suffixes` and `[CLASS PLIST CHILDREN]` for groups. At the root of the tree is an element `[N Nil CHILDREN]`, where `N` is the version of the layout format, currently and hopefully for a long time 2. While that element looks like a group vector, that element does not count when identifying a group using a coordinate vector, i.e., `[0]` is its first child, not the root element itself.

`transient-insert-suffix` *prefix loc suffix* &optional *keep-other* [Function]

`transient-append-suffix` *prefix loc suffix* &optional *keep-other* [Function]

These functions insert the suffix or group *SUFFIX* into *PREFIX* before or after *LOC*.

Conceptually adding a binding to a transient prefix is similar to adding a binding to a keymap, but this is complicated by the fact that multiple suffix commands can

be bound to the same key, provided they are never active at the same time, see Section 5.9 [Predicate Slots], page 41.

Unfortunately both false-positives and false-negatives are possible. To deal with the former, use non-`nil` *KEEP-OTHER*. The symbol `always` prevents the removal of a false-positive, in some cases where other non-`nil` values would fail. To deal with false-negatives remove the conflicting binding separately, using `transient-remove-suffix`.

`transient-replace-suffix` *prefix loc suffix* [Function]
This function replaces the suffix or group at *LOC* in *PREFIX* with suffix or group *SUFFIX*.

`transient-remove-suffix` *prefix loc* [Function]
This function removes the suffix or group at *LOC* in *PREFIX*.

`transient-get-suffix` *prefix loc* [Function]
This function returns the suffix or group at *LOC* in *PREFIX*. The returned value has the form mentioned above.

`transient-suffix-put` *prefix loc prop value* [Function]
This function edits the suffix or group at *LOC* in *PREFIX*, by setting the *PROP* of its plist to *VALUE*.

Some prefix commands share suffixes, which are separately and then included in each prefix when it is defined. The inclusion is done by reference, the included suffix groups are not inlined by default. So if you change, for example, the key binding for an argument in `magit-diff` (`d`) the same change also applies to `magit-diff-refresh` (`D`). In the rare case that this is not desirable use `transient-inline-group` before making changes to included suffixes.

`transient-inline-group` *PREFIX GROUP* [Function]
This function inlines the included GROUP into PREFIX, by replacing the symbol GROUP with its expanded layout in the layout of PREFIX.

Most of these functions do not signal an error if they cannot perform the requested modification. The functions that insert new suffixes show a warning if *LOC* cannot be found in *PREFIX* without signaling an error. The reason for doing it like this is that establishing a key binding (and that is what we essentially are trying to do here) should not prevent the rest of the configuration from loading. Among these functions only `transient-get-suffix` and `transient-suffix-put` signal an error by default. If you really want the insert functions to also signal an error, set `transient-error-on-insert-failure` to `t`.

4 Defining New Commands

4.1 Technical Introduction

Taking inspiration from prefix keys and prefix arguments, Transient implements a similar abstraction involving a prefix command, infix arguments and suffix commands.

When the user calls a transient prefix command, a transient (temporary) keymap is activated, which binds the transient's infix and suffix commands, and functions that control the transient state are added to `pre-command-hook` and `post-command-hook`. The available suffix and infix commands and their state are shown in a menu buffer until the transient state is exited by invoking a suffix command.

Calling an infix command causes its value to be changed. How that is done depends on the type of the infix command. The simplest case is an infix command that represents a command-line argument that does not take a value. Invoking such an infix command causes the switch to be toggled on or off. More complex infix commands may read a value from the user, using the minibuffer.

Calling a suffix command usually causes the transient to be exited; the transient keymaps and hook functions are removed, the menu buffer no longer shows information about the (no longer bound) suffix commands, the values of some public global variables are set, while some internal global variables are unset, and finally the command is actually called. Suffix commands can also be configured to not exit the transient.

A suffix command can, but does not have to, use the infix arguments in much the same way any command can choose to use or ignore the prefix arguments. For a suffix command that was invoked from a transient, the variable `transient-current-suffixes` and the function `transient-args` serve about the same purpose as the variables `prefix-arg` and `current-prefix-arg` do for any command that was called after the prefix arguments have been set using a command such as `universal-argument`.

Transient can be used to implement simple "command dispatchers". The main benefit then is that the user can see all the available commands in a temporarily shown buffer, which can be thought of as a "menu". That is useful by itself because it frees the user from having to remember all the keys that are valid after a certain prefix key or command. Magit's `magit-dispatch` (on *`C-x M-g`*) command is an example of using Transient to merely implement a command dispatcher.

In addition to that, Transient also allows users to interactively pass arguments to commands. These arguments can be much more complex than what is reasonable when using prefix arguments. There is a limit to how many aspects of a command can be controlled using prefix arguments. Furthermore, what a certain prefix argument means for different commands can be completely different, and users have to read documentation to learn and then commit to memory what a certain prefix argument means to a certain command.

Transient suffix commands, on the other hand, can accept dozens of different arguments without the user having to remember anything. When using Transient, one can call a command with arguments that are just as complex as when calling the same function non-interactively from Lisp.

Invoking a transient suffix command with arguments is similar to invoking a command in a shell with command-line completion and history enabled. One benefit of the Transient

interface is that it remembers history not only on a global level ("this command was invoked using these arguments, and previously it was invoked using those other arguments"), but also remembers the values of individual arguments independently. See Section 2.5 [Using History], page 5.

After a transient prefix command is invoked, `C-h KEY` can be used to show the documentation for the infix or suffix command that `KEY` is bound to (see Section 2.6 [Getting Help for Suffix Commands], page 6), and infixes and suffixes can be removed from the transient using `C-x l KEY`. Infixes and suffixes that are disabled by default can be enabled the same way. See Section 2.7 [Enabling and Disabling Suffixes], page 6.

Transient ships with support for a few different types of specialized infix commands. A command that sets a command line option, for example, has different needs than a command that merely toggles a boolean flag. Additionally, Transient provides abstractions for defining new types, which the author of Transient did not anticipate (or didn't get around to implementing yet).

Note that suffix commands also support regular prefix arguments. A suffix command may even be called with both infix and prefix arguments at the same time. If you invoke a command as a suffix of a transient prefix command, but also want to pass prefix arguments to it, then first invoke the prefix command, and only after doing that invoke the prefix arguments, before finally invoking the suffix command. If you instead began by providing the prefix arguments, then those would apply to the prefix command, not the suffix command. Likewise, if you want to change infix arguments before invoking a suffix command with prefix arguments, then change the infix arguments before invoking the prefix arguments. In other words, regular prefix arguments always apply to the next command, and since transient prefix, infix and suffix commands are just regular commands, the same applies to them. (Regular prefix keys behave differently because they are not commands at all, instead they are just incomplete key sequences, and those cannot be interrupted with prefix commands.)

4.2 Defining Transients

A transient consists of a prefix command and at least one suffix command, though usually a transient has several infix and suffix commands. The below macro defines the transient prefix command **and** binds the transient's infix and suffix commands. In other words, it defines the complete transient, not just the transient prefix command that is used to invoke that transient.

`transient-define-prefix` *name arglist* [*docstring*] [*keyword value*]... *group*... [*body*...] [Macro]

This macro defines *NAME* as a transient prefix command and binds the transient's infix and suffix commands.

ARGLIST are the arguments that the prefix command takes. *DOCSTRING* is the documentation string and is optional.

These arguments can optionally be followed by keyword-value pairs. Each key has to be a keyword symbol, either `:class` or a keyword argument supported by the constructor of that class. The `transient-prefix` class is used if the class is not specified explicitly.

*GROUP*s add key bindings for infix and suffix commands and specify how these bindings are presented in the menu buffer. At least one *GROUP* has to be specified. See Section 4.3 [Binding Suffix and Infix Commands], page 18.

The *BODY* is optional. If it is omitted, then *ARGLIST* is ignored and the function definition becomes:

```
(lambda ()
  (interactive)
  (transient-setup 'NAME))
```

If *BODY* is specified, then it must begin with an `interactive` form that matches *ARGLIST*, and it must call `transient-setup`. It may, however, call that function only when some condition is satisfied.

All transients have a (possibly `nil`) value, which is exported when suffix commands are called, so that they can consume that value. For some transients it might be necessary to have a sort of secondary value, called a "scope". Such a scope would usually be set in the command's `interactive` form and has to be passed to the setup function:

```
(transient-setup 'NAME nil nil :scope SCOPE)
```

For example, the scope of the `magit-branch-configure` transient is the branch whose variables are being configured.

Sometimes multiple prefix commands share a common set of suffixes. For example, while `magit-diff` (`d`) and `magit-diff-refresh` (`D`) offer different suffixes to actually create or update a diff, they both offer the same infix arguments to control how that diff is formatted. Such shared groups should be defined using `transient-define-group` and then included in multiple prefixes, by using the symbol that identifies the group in the prefix definition, in a location where you would otherwise use a group vector. If an included group is placed at the top-level of a prefix (as opposed of inside inside a vector as a child group), then the symbol should be quoted.

`transient-define-group` *name group...* [Macro]

This macro define one or more groups and stores them in symbol NAME. GROUPs have the same form as for `transient-define-prefix`.

4.3 Binding Suffix and Infix Commands

The macro `transient-define-prefix` is used to define a transient. This defines the actual transient prefix command (see Section 4.2 [Defining Transients], page 17) and adds the transient's infix and suffix bindings, as described below.

Users and third-party packages can add additional bindings using functions such as `transient-insert-suffix` (see Chapter 3 [Modifying Existing Transients], page 14). These functions take a "suffix specification" as one of their arguments, which has the same form as the specifications used in `transient-define-prefix`.

4.3.1 Group Specifications

The suffix and infix commands of a transient are organized in groups. The grouping controls how the descriptions of the suffixes are outlined visually but also makes it possible to set certain properties for a set of suffixes.

Several group classes exist, some of which organize suffixes in subgroups. In most cases the class does not have to be specified explicitly, but see Section 5.1 [Group Classes], page 30.

Groups are specified in the call to `transient-define-prefix`, using vectors. Because groups are represented using vectors, we cannot use square brackets to indicate an optional element and instead use curly brackets to do the latter.

Group specifications then have this form:

```
[{LEVEL} {DESCRIPTION} {KEYWORD VALUE}... ELEMENT...]
```

The *LEVEL* is optional and defaults to 4. See Section 2.7 [Enabling and Disabling Suffixes], page 6.

The *DESCRIPTION* is optional. If present, it is used as the heading of the group.

The *KEYWORD-VALUE* pairs are optional. Each keyword has to be a keyword symbol, either `:class` or a keyword argument supported by the constructor of that class.

- One of these keywords, `:description`, is equivalent to specifying *DESCRIPTION* at the very beginning of the vector. The recommendation is to use `:description` if some other keyword is also used, for consistency, or *DESCRIPTION* otherwise, because it looks better.
- Likewise `:level` is equivalent to *LEVEL*.
- Other important keywords include the `:if...` and `:inapt-if...` keywords. These keywords control whether the group is available in a certain situation.

  For example, one group of the `magit-rebase` transient uses `:if magit-rebase-in-progress-p`, which contains the suffixes that are useful while rebase is already in progress; and another that uses `:if-not magit-rebase-in-progress-p`, which contains the suffixes that initiate a rebase.

  These predicates can also be used on individual suffixes and are only documented once, see Section 5.9 [Predicate Slots], page 41.
- The value of `:hide`, if non-`nil`, is a predicate that controls whether the group is hidden by default. The key bindings for suffixes of a hidden group should all use the same prefix key. Pressing that prefix key should temporarily show the group and its suffixes, which assumes that a predicate like this is used:

  ```
  (lambda ()
    (eq (car transient--redisplay-key)
        ?\C-c)) ; the prefix key shared by all bindings
  ```
- The value of `:setup-children`, if non-`nil`, is a function that takes one argument, a potentially list of children, and must return a list of children or an empty list. This can either be used to somehow transform the group's children that were defined the normal way, or to dynamically create the children from scratch.

  The returned children must have the same form as stored in the prefix's `transient--layout` property, but it is often more convenient to use the same form as understood by `transient-define-prefix`, described below. If you use the latter approach, you can use the `transient-parse-suffixes` and `transient-parse-suffix` functions to transform them from the convenient to the expected form. Depending on the used group class, `transient-parse-suffixes`'s SUFFIXES must be a list of group vectors (for `transient-columns`) or a list of suffix lists (for all other group classes).

If you explicitly specify children and then transform them using `:setup-children`, then the class of the group is determined as usual, based on explicitly specified children.

If you do not explicitly specify children and thus rely solely on `:setup-children`, then you must specify the class using `:class`. For backward compatibility, if you fail to do so, `transient-column` is used and a warning is displayed. This warning will eventually be replaced with an error.

```
(transient-define-prefix my-finder-by-keyword ()
  "Select a keyword and list matching packages."
  ;; The real `finder-by-keyword' is more convenient
  ;; of course, but that is not the point here.
  [:class transient-columns
   :setup-children
   (lambda (_)
     (transient-parse-suffixes
      'my-finder-by-keyword
      (let ((char (1- ?A)))
        (mapcar                  ; a list ...
         (lambda (partition)
           (vconcat              ; of group vectors ...
            (mapcar (lambda (elt)
                      (let ((keyword (symbol-name (car elt))))
                                  ; ... where each suffix is a list
                        (list (format "%c" (cl-incf char))
                              keyword
                              (lambda ()
                                (interactive)
                                (finder-list-matches keyword)))))
                    partition)))
         (seq-partition finder-known-keywords 7)))))])
```

- The boolean `:pad-keys` argument controls whether keys of all suffixes contained in a group are right padded, effectively aligning the descriptions.
- If a keyword argument accepts a function as value, you an use a `lambda` expression. As a special case, the `##` macro (which returns a `lambda` expression and is implemented in the `llama` package) is also supported. Inside group specifications, the use of `##` is not supported anywhere but directly following a keyword symbol.

The *ELEMENT*s are either all subgroups, or all suffixes and strings. (At least currently no group type exists that would allow mixing subgroups with commands at the same level, though in principle there is nothing that prevents that.)

If the *ELEMENT*s are not subgroups, then they can be a mixture of lists, which specify commands, and strings. Strings are inserted verbatim into the buffer. The empty string can be used to insert gaps between suffixes, which is particularly useful if the suffixes are outlined as a table.

Inside group specifications, including inside contained suffix specifications, nothing has to be quoted and quoting anyway is invalid. The value following a keyword, can be explicitly unquoted using `,`. This feature is experimental and should be avoided.

The form of suffix specifications is documented in the next node.

4.3.2 Suffix Specifications

A transient's suffix and infix commands are bound when the transient prefix command is defined using `transient-define-prefix`, see Section 4.2 [Defining Transients], page 17. The commands are organized into groups, see Section 4.3.1 [Group Specifications], page 18. Here we describe the form used to bind an individual suffix command.

The same form is also used when later binding additional commands using functions such as `transient-insert-suffix`, see Chapter 3 [Modifying Existing Transients], page 14.

Note that an infix is a special kind of suffix. Depending on context "suffixes" means "suffixes (including infixes)" or "non-infix suffixes". Here it means the former.

Suffix specifications have this form:

```
([LEVEL] [KEY [DESCRIPTION]] COMMAND|ARGUMENT [KEYWORD VALUE]...)
```

LEVEL, *KEY* and *DESCRIPTION* can also be specified using the *KEYWORD*s `:level`, `:key` and `:description`. If the object that is associated with *COMMAND* sets these properties, then they do not have to be specified here. You can however specify them here anyway, possibly overriding the object's values just for the binding inside this transient.

- *LEVEL* is the suffix level, an integer between 1 and 7. See Section 2.7 [Enabling and Disabling Suffixes], page 6.
- KEY is the key binding, a string in the format returned by `describe-key` and understood by `kbd`.

  That format is more permissive than the one accepted by `key-valid-p`. Being more permissive makes it possible, for example, to write the key binding, which toggles the `-a` command line argument, as `"-a"`, instead of having to write `"- a"`. Likewise additional spaces can be added, which is not removed when displaying the binding in the menu, which is useful for alignment purposes.
- *DESCRIPTION* is the description, either a string or a function that takes zero or one arguments (the suffix object) and returns a string. The function should be a lambda expression to avoid ambiguity. In some cases a symbol that is bound as a function would also work but to be safe you should use `:description` in that case.

The next element is either a command or an argument. This is the only argument that is mandatory in all cases.

- *COMMAND* should be a symbol that is bound as a function, which has to be defined or at least autoloaded as a command by the time the containing prefix command is invoked.

  Any command will do; it does not need to have an object associated with it (as would be the case if `transient-define-suffix` or `transient-define-infix` were used to define it).

  COMMAND can also be a `lambda` expression.

  As mentioned above, the object that is associated with a command can be used to set the default for certain values that otherwise have to be set in the suffix specification. Therefore if there is no object, then you have to make sure to specify the *KEY* and the *DESCRIPTION*.

As a special case, if you want to add a command that might be neither defined nor autoloaded, you can use a workaround like:

```
(transient-insert-suffix 'some-prefix "k"
  '("!" "Ceci n'est pas une commande" no-command
    :if (lambda () (featurep 'no-library))))
```

Instead of `featurep` you could also use `require` with a non-`nil` value for *NOERROR*.

- The mandatory argument can also be a command-line argument, a string. In that case an anonymous command is defined and bound.

  Instead of a string, this can also be a list of two strings, in which case the first string is used as the short argument (which can also be specified using `:shortarg`) and the second as the long argument (which can also be specified using `:argument`).

  Only the long argument is displayed in the menu buffer. See `transient-detect-key-conflicts` for how the short argument may be used.

  Unless the class is specified explicitly, the appropriate class is guessed based on the long argument. If the argument ends with '`=`' (e.g., '`--format=`') then `transient-option` is used, otherwise `transient-switch`.

Finally, details can be specified using optional *KEYWORD-VALUE* pairs. Each keyword has to be a keyword symbol, either `:class` or a keyword argument supported by the constructor of that class. See Section 5.8 [Suffix Slots], page 38.

If a keyword argument accepts a function as value, you an use a `lambda` expression. As a special case, the `##` macro (which returns a `lambda` expression and is implemented in the `llama` package) is also supported. Inside suffix bindings, the use of `##` is not supported anywhere but directly following a keyword symbol.

4.4 Defining Suffix and Infix Commands

Note that an infix is a special kind of suffix. Depending on context "suffixes" means "suffixes (including infixes)" or "non-infix suffixes".

`transient-define-suffix` *name arglist* [*docstring*] [*keyword value*]... *body...* [Macro]

This macro defines *NAME* as a transient suffix command.

ARGLIST are the arguments that the command takes. *DOCSTRING* is the documentation string and is optional.

These arguments can optionally be followed by keyword-value pairs. Each keyword has to be a keyword symbol, either `:class` or a keyword argument supported by the constructor of that class. The `transient-suffix` class is used if the class is not specified explicitly.

The *BODY* must begin with an `interactive` form that matches *ARGLIST*. The infix arguments are usually accessed by using `transient-args` inside `interactive`.

`transient-define-infix` *name arglist* [*docstring*] [*keyword value*]... [Macro]

This macro defines *NAME* as a transient infix command.

ARGLIST is always ignored (but mandatory never-the-less) and reserved for future use. *DOCSTRING* is the documentation string and is optional.

At least one key-value pair is required. All transient infix commands are `equal` to each other (but not `eq`). It is meaningless to define an infix command, without providing at least one keyword argument (usually `:argument` or `:variable`, depending on the class). The suffix class defaults to `transient-switch` and can be set using the `:class` keyword.

The function definition is always:

```
(lambda ()
  (interactive)
  (let ((obj (transient-suffix-object)))
    (transient-infix-set obj (transient-infix-read obj)))
  (transient--show))
```

`transient-infix-read` and `transient-infix-set` are generic functions. Different infix commands behave differently because the concrete methods are different for different infix command classes. In rare cases the above command function might not be suitable, even if you define your own infix command class. In that case you have to use `transient-define-suffix` to define the infix command and use `t` as the value of the `:transient` keyword.

`transient-define-argument` *name arglist* [*docstring*] [*keyword value*]... [Macro]

This macro defines *NAME* as a transient infix command.

This is an alias for `transient-define-infix`. Only use this alias to define an infix command that actually sets an infix argument. To define an infix command that, for example, sets a variable, use `transient-define-infix` instead.

4.5 Using Infix Arguments

The functions and the variables described below allow suffix commands to access the value of the transient from which they were invoked; which is the value of its infix arguments. These variables are set when the user invokes a suffix command that exits the transient, but before actually calling the command.

When returning to the command-loop after calling the suffix command, the arguments are reset to `nil` (which causes the function to return `nil` too).

Like for Emacs's prefix arguments, it is advisable, but not mandatory, to access the infix arguments inside the command's `interactive` form. The preferred way of doing that is to call the `transient-args` function, which for infix arguments serves about the same purpose as `prefix-arg` serves for prefix arguments.

`transient-args` *prefix* [Function]

This function returns the value of the transient prefix command *PREFIX*.

If the current command was invoked from the transient prefix command *PREFIX*, then it returns the active infix arguments. If the current command was not invoked from *PREFIX*, then it returns the set, saved or default value for *PREFIX*.

PREFIX may also be a list of prefixes. If no prefix is active, the fallback value of the first of these prefixes is used.

The generic function `transient-prefix-value` is used to determine the returned value.

This function is intended to be used by suffix commands, whether they are invoked from a menu or not. It is not intended to be used when setting up a menu and its suffixes, in which case `transient-get-value` should be used.

`transient-get-value` [Function]

This function returns the value of the erant prefix.

This function is intended to be used when setting up a menu and its suffixes. It is not intended to be used when a suffix command is invoked, whether from a menu or not, in which case `transient-args` should be used. In other words, use this, e.g., in a suffixes `:if*` or `:inapt-if*` predicate and `:description` function, but never in its `interactive` form or function body.

`transient-arg-value` *arg args* [Function]

This function returns the value of *ARG* as it appears in *ARGS*.

For a switch a boolean is returned. For an option the value is returned as a string, using the empty string for the empty value, or `nil` if the option does not appear in *ARGS*.

`transient-suffixes` *prefix* [Function]

This function returns the suffixes of the transient prefix command *PREFIX*. This is a list of objects. This function should only be used if you need the objects (as opposed to just their values) and if the current command is not being invoked from *PREFIX*.

4.6 Using Prefix Scope

Some transients have a sort of secondary value, called a scope. A prefix's scope can be accessed using `transient-scope`; similar to how its value can be accessed using `transient-args`.

`transient-scope` *prefixes classes* [Function]

This function returns the scope of the active or current transient prefix command.

If optional PREFIXES and CLASSES are both `nil`, return the scope of the prefix currently being setup, making this variation useful, e.g., in `:if*` predicates. If no prefix is being setup, but the current command was invoked from some prefix, then return the scope of that.

If PREFIXES is non-`nil`, it must be a prefix command or a list of such commands. If CLASSES is non-`nil`, it must be a prefix class or a list of such classes. When this function is called from the body or the `interactive` form of a suffix command, PREFIXES and/or CLASSES should be non-`nil`. If either is non-`nil`, try the following in order:

- If the current suffix command was invoked from a prefix, which appears in PREFIXES, return the scope of that prefix.
- If the current suffix command was invoked from a prefix, and its class derives from one of the CLASSES, return the scope of that prefix.
- If a prefix is being setup and it appears in PREFIXES, return its scope.

- If a prefix is being setup and its class derives from one of the CLASSES, return its scope.
- Finally try to return the default scope of the first command in PREFIXES. This only works if that slot is set in the respective class definition or using its 'transient-init-scope' method.

If no prefix matches, return `nil`.

4.7 Current Suffix Command

`transient-suffix-object` *command* [Function]

This function returns the object associated with the current suffix command.

Each suffix commands is associated with an object, which holds additional information about the suffix, such as its value (in the case of an infix command, which is a kind of suffix command).

This function is intended to be called by infix commands, which are usually aliases of `transient--default-infix-command`, which is defined like this:

```
(defun transient--default-infix-command ()
  (interactive)
  (let ((obj (transient-suffix-object)))
    (transient-infix-set obj (transient-infix-read obj)))
  (transient--show))
```

(User input is read outside of `interactive` to prevent the command from being added to `command-history`.)

Such commands need to be able to access their associated object to guide how `transient-infix-read` reads the new value and to store the read value. Other suffix commands (including non-infix commands) may also need the object to guide their behavior.

This function attempts to return the object associated with the current suffix command even if the suffix command was not invoked from a transient. (For some suffix command that is a valid thing to do, for others it is not.) In that case `nil` may be returned, if the command was not defined using one of the macros intended to define such commands.

The optional argument COMMAND is intended for internal use. If you are contemplating using it in your own code, then you should probably use this instead:

```
(get COMMAND 'transient--suffix)
```

`transient-current-suffixes` [Variable]

The suffixes of the transient from which this suffix command was invoked. This is a list of objects. Usually it is sufficient to instead use the function `transient-args`, which returns a list of values. In complex cases it might be necessary to use this variable instead, i.e., if you need access to information beside the value.

4.8 Current Prefix Command

`transient-prefix-object` [Function]

This function returns the current prefix as an object.

While a transient is being setup or refreshed (which involves preparing its suffixes) the variable `transient--prefix` can be used to access the prefix object. Thus this is what has to be used in suffix methods such as `transient-format-description`, and in object-specific functions that are stored in suffix slots such as `description`.

When a suffix command is invoked (i.e., in its `interactive` form and function body) then the variable `transient-current-prefix` has to be used instead.

Two distinct variables are needed, because any prefix may itself be used as a suffix of another prefix, and such sub-prefixes have to be able to tell themselves apart from the prefix they were invoked from.

Regular suffix commands, which are not prefixes, do not have to concern themselves with this distinction, so they can use this function instead. In the context of a plain suffix, it always returns the value of the appropriate variable.

`transient-current-prefix` [Variable]

The transient from which this suffix command was invoked. The value is a `transient-prefix` object, which holds information associated with the transient prefix command.

`transient-current-command` [Variable]

The transient from which this suffix command was invoked. The value is a symbol, the transient prefix command.

`transient-active-prefix` &optional *prefixes* [Function]

This function returns the active transient object. It returns `nil` if there is no active transient, if the transient buffer isn't shown, and while the active transient is suspended (e.g., while the minibuffer is in use).

Unlike `transient-current-prefix`, which is only ever non-`nil` in code that is run directly by a command that is invoked while a transient is current, this function is also suitable for use in asynchronous code, such as timers and callbacks (this function's main use-case).

If optional PREFIXES is non-`nil`, it must be a prefix command symbol or a list of symbols, in which case the active transient object is only returned if it matches one of the PREFIXES.

4.9 Transient State

Invoking a transient prefix command "activates" the respective transient, i.e., it puts a transient keymap into effect, which binds the transient's infix and suffix commands.

The default behavior while a transient is active is as follows:

- Invoking an infix command does not affect the transient state; the transient remains active.
- Invoking a (non-infix) suffix command "deactivates" the transient state by removing the transient keymap and performing some additional cleanup.

- Invoking a command that is bound in a keymap other than the transient keymap is disallowed and trying to do so results in a warning. This does not "deactivate" the transient.

The behavior can be changed for all suffixes of a particular prefix and/or for individual suffixes. The values should nearly always be booleans, but certain functions, called "pre-commands", can also be used. These functions are named `transient--do-VERB`, and the symbol `VERB` can be used as a shorthand.

A boolean is interpreted as answering the question "does the transient stay active, when this command is invoked?" `t` means that the transient stays active, while `nil` means that invoking the command exits the transient.

Note that when the suffix is a "sub-prefix", invoking that command always activates that sub-prefix, causing the outer prefix to no longer be active and displayed. Here `t` means that when you exit the inner prefix, then the outer prefix becomes active again, while `nil` means that all outer prefixes are exited at once.

- The behavior for non-suffixes can be set for a particular prefix, by the prefix's `transient-non-suffix` slot to a boolean, a suitable pre-command function, or a shorthand for such a function. See [Pre-commands for Non-Suffixes], page 29.
- The common behavior for the suffixes of a particular prefix can be set using the prefix's `transient-suffixes` slot.

  The value specified in this slot does **not** affect infixes. Because it affects both regular suffixes as well as sub-prefixes, which have different needs, it is best to avoid explicitly specifying a function.
- The behavior of an individual suffix can be changed using its `transient` slot. While it is usually best to use a boolean, for this slot it can occasionally make sense to specify a function explicitly.

  Note that this slot can be set when defining a suffix command using `transient-define-suffix` and/or in the definition of the prefix. If set in both places, then the latter takes precedence, as usual.

The available pre-command functions are documented in the following sub-sections. They are called by `transient--pre-command`, a function on `pre-command-hook`, and the value that they return determines whether the transient is exited. To do so the value of one of the constants `transient--exit` or `transient--stay` is used (that way we don't have to remember if `t` means "exit" or "stay").

Additionally, these functions may change the value of `this-command` (which explains why they have to be called using `pre-command-hook`), call `transient-export`, `transient--stack-zap` or `transient--stack-push`; and set the values of `transient--exitp`, `transient--helpp` or `transient--editp`.

For completeness sake, some notes about complications:

- The transient-ness of certain built-in suffix commands is specified using `transient-predicate-map`. This is a special keymap, which binds commands to pre-commands (as opposed to keys to commands) and takes precedence over the prefix's `transient-suffix` slot, but not the suffix's `transient` slot.
- While a sub-prefix is active we nearly always want `C-g` to take the user back to the "super-prefix", even when the other suffixes don't do that. However, in rare cases this

may not be desirable, in which case `replace` can be used as the value of the sub-prefix's `transient` slot.

Pre-commands for Infixes

The default for infixes is `transient--do-stay`. This is also the only function that makes sense for infixes, which is why this predicate is used even if the value of the prefix's `transient-suffix` slot is `t`. In extremely rare cases, one might want to use something else, which can be done by setting the infix's `transient` slot directly.

`transient--do-stay` [Function]
Call the command without exporting variables and stay transient.

Pre-commands for Suffixes

By default, invoking a suffix causes the transient to be exited.

The behavior for an individual suffix command can be changed by setting its `transient` slot to a boolean (which is highly recommended), or to one of the following pre-commands.

`transient--do-exit` [Function]
Call the command after exporting variables and exit the transient.

`transient--do-return` [Function]
Call the command after exporting variables and return to the parent prefix. If there is no parent prefix, then call `transient--do-exit`.

`transient--do-call` [Function]
Call the command after exporting variables and stay transient.

The following pre-commands are only suitable for sub-prefixes. It is not necessary to explicitly use these predicates because the correct predicate is automatically picked based on the value of the `transient` slot for the sub-prefix itself.

`transient--do-recurse` [Function]
Call the transient prefix command, preparing for return to outer transient.

Whether we actually return to the parent transient is ultimately under the control of each invoked suffix. The difference between this pre-command and `transient--do-stack` is that it changes the value of the `transient-suffix` slot to `t`.

If there is no parent transient, then only call this command and skip the second step.

`transient--do-stack` [Function]
Call the transient prefix command, stacking the active transient. Push the active transient to the transient stack.

Unless `transient--do-recurse` is explicitly used, this pre-command is automatically used for suffixes that are prefixes themselves, i.e., for sub-prefixes.

`transient--do-replace` [Function]
Call the transient prefix command, replacing the active transient. Do not push the active transient to the transient stack.

Unless `transient--do-recurse` is explicitly used, this pre-command is automatically used for suffixes that are prefixes themselves, i.e., for sub-prefixes.

`transient--do-suspend` [Function]

Suspend the active transient, saving the transient stack.

This is used by the command `transient-suspend` and optionally also by "external events" such as `handle-switch-frame`. Such bindings should be added to `transient-predicate-map`.

Pre-commands for Non-Suffixes

By default, non-suffixes (commands that are bound in other keymaps beside the transient keymap) cannot be invoked. Trying to invoke such a command results in a warning and the transient stays active.

If you want a different behavior, then set the `transient-non-suffix` slot of the transient prefix command. The value should be a boolean, answering the question, "is it allowed to invoke non-suffix commands?, a pre-command function, or a shorthand for such a function.

If the value is `t`, then non-suffixes can be invoked, when it is `nil` (the default) then they cannot be invoked.

The only other recommended value is `leave`. If that is used, then non-suffixes can be invoked, but if one is invoked, then that exits the transient.

`transient--do-warn` [Function]

Call `transient-undefined` and stay transient.

`transient--do-stay` [Function]

Call the command without exporting variables and stay transient.

`transient--do-leave` [Function]

Call the command without exporting variables and exit the transient.

Special Pre-Commands

`transient--do-quit-one` [Function]

If active, quit help or edit mode, else exit the active transient.

This is used when the user pressed *C-g*.

`transient--do-quit-all` [Function]

Exit all transients without saving the transient stack.

This is used when the user pressed *C-q*.

`transient--do-suspend` [Function]

Suspend the active transient, saving the transient stack.

This is used when the user pressed *C-z*.

5 Classes and Methods

Transient uses classes and generic functions to make it possible to define new types of suffix and prefix commands, which are similar to existing types, but behave differently in some respects.

Every prefix, infix and suffix command is associated with an object, which holds information, which controls certain aspects of its behavior. This happens in two ways.

- Associating a command with a certain class gives the command a type. This makes it possible to use generic functions to do certain things that have to be done differently depending on what type of command it acts on.

  That in turn makes it possible for third-parties to add new types without having to convince the maintainer of Transient, that that new type is important enough to justify adding a special case to a dozen or so functions.

- Associating a command with an object makes it possible to easily store information that is specific to that particular command.

  Two commands may have the same type, but obviously their key bindings and descriptions still have to be different, for example.

  The values of some slots are functions. The `reader` slot for example holds a function that is used to read a new value for an infix command. The values of such slots are regular functions.

  Generic functions are used when a function should do something different based on the type of the command, i.e., when all commands of a certain type should behave the same way but different from the behavior for other types. Object slots that hold a regular function as value are used when the task that they perform is likely to differ even between different commands of the same type.

5.1 Group Classes

The type of a group can be specified using the `:class` property at the beginning of the class specification, e.g., `[:class transient-columns ...]` in a call to `transient-define-prefix`.

- The abstract `transient-child` class is the base class of both `transient-group` (and therefore all groups) as well as of `transient-suffix` (and therefore all suffix and infix commands).

  This class exists because the elements (or "children") of certain groups can be other groups instead of suffix and infix commands.

- The abstract `transient-group` class is the superclass of all other group classes.

- The `transient-column` class is the simplest group.

  This is the default "flat" group. If the class is not specified explicitly and the first element is not a vector (i.e., not a group), then this class is used.

  This class displays each element on a separate line.

- The `transient-row` class displays all elements on a single line.

- The `transient-columns` class displays commands organized in columns.

  Direct elements have to be groups whose elements have to be commands or strings. Each subgroup represents a column. This class takes care of inserting the subgroups' elements.

  This is the default "nested" group. If the class is not specified explicitly and the first element is a vector (i.e., a group), then this class is used.

- The `transient-subgroups` class wraps other groups.

  Direct elements have to be groups whose elements have to be commands or strings. This group inserts an empty line between subgroups. The subgroups themselves are responsible for displaying their elements.

5.2 Group Methods

`transient-setup-children` *group children* [Function]

This generic function can be used to setup the children or a group.

The default implementation usually just returns the children unchanged, but if the `setup-children` slot of *GROUP* is non-`nil`, then it calls that function with *CHILDREN* as the only argument and returns the value.

The children are given as a (potentially empty) list consisting of either group or suffix specifications. These functions can make arbitrary changes to the children including constructing new children from scratch.

`transient--insert-group` *group* [Function]

This generic function formats the group and its elements and inserts the result into the current buffer, which is a temporary buffer. The contents of that buffer are later inserted into the menu buffer.

Functions that are called by this function may need to operate in the buffer from which the transient was called. To do so they can temporarily make the `transient--shadowed-buffer` the current buffer.

5.3 Prefix Classes

Transient itself provides a single class for prefix commands, `transient-prefix`, but package authors may wish to define specialized classes. Doing so makes it possible to change the behavior of the set of prefix commands that use that class, by implementing specialized methods for certain generic functions (see Section 5.5 [Prefix Methods], page 33).

A transient prefix command's object is stored in the `transient--prefix` property of the command symbol. While a transient is active, a clone of that object is stored in the variable `transient--prefix`. A clone is used because some changes that are made to the active transient's object should not affect later invocations.

5.4 Suffix Classes

- All suffix and infix classes derive from `transient-suffix`, which in turn derives from `transient-child`, from which `transient-group` also derives (see Section 5.1 [Group Classes], page 30).

- All infix classes derive from the abstract `transient-infix` class, which in turn derives from the `transient-suffix` class.

  Infixes are a special type of suffixes. The primary difference is that infixes always use the `transient--do-stay` pre-command, while non-infix suffixes use a variety of pre-commands (see Section 4.9 [Transient State], page 26). Doing that is most easily achieved by using this class, though theoretically it would be possible to define an infix class that does not do so. If you do that then you get to implement many methods.

  Also, infixes and non-infix suffixes are usually defined using different macros (see Section 4.4 [Defining Suffix and Infix Commands], page 22).
- Classes used for infix commands that represent arguments should be derived from the abstract `transient-argument` class.
- The `transient-switch` class (or a derived class) is used for infix arguments that represent command-line switches (arguments that do not take a value).
- The `transient-option` class (or a derived class) is used for infix arguments that represent command-line options (arguments that do take a value).
- The `transient-switches` class can be used for a set of mutually exclusive command-line switches.
- The `transient-files` class can be used for a '`--`' argument that indicates that all remaining arguments are files.
- Classes used for infix commands that represent variables should derived from the abstract `transient-variable` class.
- The `transient-information` and `transient-information*` classes are special in that suffixes that use these class are not associated with a command and thus also not with any key binding. Such suffixes are only used to display arbitrary information, and that anywhere a suffix can appear. Display-only suffix specifications take these form:

  ```
  (:info DESCRIPTION [KEYWORD VALUE]...)
  (:info* DESCRIPTION [KEYWORD VALUE]...)
  ```

  The `:info` and `:info*` keyword arguments replaces the `:description` keyword used for other suffix classes. Other keyword arguments that you might want to set, include `:face`, predicate keywords (such as `:if` and `:inapt-if`), and `:format`. By default the value of `:format` includes `%k`, which for this class is replaced with the empty string or spaces, if keys are being padded in the containing group.

  The only difference between these two classes is that `:info*` aligns its description with the descriptions of suffix commands, while for `:info` the description bleeds into the area where suffixes display their key bindings.
- The `transient-lisp-variable` class can be used to show and change the value of lisp variables. This class is not fully featured yet and it is somewhat likely that future improvements won't be fully backward compatible.
- The `transient-cons-option` class is intended for situations where `transient-args` should return an alist, instead of a list of strings (arguments). Such suffixes can be specified in prefix definitions like so:

  ```
  (:cons OPTION :key KEY [KEYWORD VALUE]...)
  ```

  OPTION may be something other than a string, likely a keyword or some other symbol, it is used as the `car` of the cons-cell. When using such an inline definition `:key` has

to be specified. In most cases `:reader` should also be specified. When defining such a suffix separately, the "alist key" has to be specified using the `:variable` keyword argument.

This class is still experimental it is somewhat likely that future improvements won't be fully backward compatible.

- The `transient-describe-target` class is used by the command `transient-describe`.
- The `transient-value-preset` class is used to implement the command `transient-preset`, which activates a value preset.

Magit defines additional classes, which can serve as examples for the fancy things you can do without modifying Transient. Some of these classes will likely get generalized and added to Transient. For now they are very much subject to change and not documented.

5.5 Prefix Methods

To get information about the methods implementing these generic functions use `describe-function`.

`transient-init-value` *obj* [Function]

This generic function sets the initial value of the object *OBJ*. Methods exist for both prefix and suffix objects.

The default method for prefix objects sets the value of OBJ's `value` slot to the set, saved or default value. The value that is set for the current session is preferred over the saved value, which is preferred over the default value.

The default value is determined using the generic function `transient-default-value`. If you need to change how the value for a prefix class is determined, its usually sufficient to implement a method for that function.

`transient-default-value` *obj* [Function]

This generic function returns the default value of the object *OBJ*. Methods exist for both prefix and suffix objects.

The default method for prefix objects returns the value of the `default-value` slot if that is bound and not a function. If it is a function, that is called to get the value. If the slot is unbound, `nil` is returned.

`transient-prefix-value` *obj* [Function]

This generic function returns the value of the prefix object *OBJ*.

OBJ is a prototype object and is only used to select the appropriate method of this generic function. This function does not return the value of that object. Instead it extracts the name of the respective command from the object and uses that to collect the current values from the suffixes of the prefix from which the current command was invoked. If the current command was not invoked from the identified prefix, then this method returns the set, save or default value, as described for `transient-args`.

This function is only intended to be used by `transient-args`. It is not defined as an internal function because third-party packages may define their own methods. That does not mean that it would be a good idea to call it for any other purpose.

The respective generic function for infix and suffix objects is named `transient-infix-value`.

`transient-init-scope` *obj* [Function]
: This generic function sets the scope of the object *OBJ*. Methods exist for both prefix and suffix objects.

  This function is called for all prefix and suffix commands, but unless a concrete method is implemented this falls through to the default implementation, which is a noop.

`transient-set-value`, `transient-save-value`, `transient-reset-value`, `transient--history-key`, `transient--history-push` and `transient--history-init` are other generic functions dealing with the value of prefix objects. See their doc-strings for more information.

`transient-show-help` is another generic function implemented for prefix commands. The default method effectively describes the command using `describe-function`.

5.6 Suffix Methods

To get information about the methods implementing these generic functions use `describe-function`.

5.6.1 Suffix Value Methods

`transient-init-value` *obj* [Function]
: This generic function sets the initial value of the object *OBJ*. Methods exist for both prefix and suffix objects.

  For `transient-argument` objects this function handles setting the value by itself.

  For other `transient-suffix` objects (including `transient-infix` objects), this calls `transient-default-value` and uses the value returned by that, unless it is the special value `eieio--unbound`, which indicates that there is no default value. Since that is what the default method for `transient-suffix` objects does, both of these functions effectively are noops for these classes.

  If you implement a class that derives from `transient-infix` directly, then you must implement a dedicated method for this function and/or `transient-default-value`.

`transient-default-value` *obj* [Function]
: This generic function returns the default value of the object *OBJ*. Methods exist for both prefix and suffix objects.

`transient-infix-read` *obj* [Function]
: This generic function determines the new value of the infix object *OBJ*.

  This function merely determines the value; `transient-infix-set` is used to actually store the new value in the object.

  For most infix classes this is done by reading a value from the user using the reader specified by the `reader` slot (using the `transient-infix-value` method described below).

For some infix classes the value is changed without reading anything in the minibuffer, i.e., the mere act of invoking the infix command determines what the new value should be, based on the previous value.

`transient-prompt` *obj* [Function]
This generic function returns the prompt to be used to read infix object *OBJ*'s value.

`transient-infix-set` *obj value* [Function]
This generic function sets the value of infix object *OBJ* to *VALUE*.

`transient-infix-value` *obj* [Function]
This generic function returns the value of the suffix object *OBJ*.

This function is called by `transient-args` (which see), meaning this function is how the value of a transient is determined so that the invoked suffix command can use it.

Currently most values are strings, but that is not set in stone. `nil` is not a value, it means "no value".

Usually only infixes have a value, but see the method for `transient-suffix`.

`transient-init-scope` *obj* [Function]
This generic function sets the scope of the object *OBJ*. Methods exist for both prefix and suffix objects.

The scope is actually a property of the transient prefix, not of individual suffixes. However it is possible to invoke a suffix command directly instead of from a transient. In that case, if the suffix expects a scope, then it has to determine that itself and store it in its `scope` slot.

This function is called for all prefix and suffix commands, but unless a concrete method is implemented, this falls through to the default implementation, which is a noop.

5.6.2 Suffix Format Methods

`transient-format` *obj* [Function]
This generic function formats and returns *OBJ* for display.

When this function is called, then the current buffer is some temporary buffer. If you need the buffer from which the prefix command was invoked to be current, then do so by temporarily making `transient--source-buffer` current.

`transient-format-key` *obj* [Function]
This generic function formats *OBJ*'s `key` for display and returns the result.

`transient-format-description` *obj* [Function]
This generic function formats *OBJ*'s `description` for display and returns the result.

`transient-format-value` *obj* [Function]
This generic function formats *OBJ*'s value for display and returns the result.

`transient-show-help` *obj* [Function]

Show help for the prefix, infix or suffix command represented by *OBJ*.

Regardless of OBJ's type, if its `show-help` slot is non-`nil`, that must be a function, which takes OBJ is its only argument. It must prepare, display and return a buffer, and select the window used to display it. The `transient-show-help-window` macro is intended for use in such functions.

For prefixes, show the info manual, if that is specified using the `info-manual` slot. Otherwise, show the manpage if that is specified using the `man-page` slot. Otherwise, show the command's documentation string.

For suffixes, show the command's documentation string.

For infixes, show the manpage if that is specified. Otherwise show the command's documentation string.

`transient-with-help-window` &rest *body* [Macro]

Evaluate BODY, send output to `*Help*` buffer, and display it in a window. Select the help window, and make the help buffer current and return it.

`transient-show-summary` *obj* &optional *return* [Function]

This generic function shows or, if optional RETURN is non-`nil`, returns a brief summary about the command at point or hovered with the mouse.

This function is called when the mouse is moved over a command and (if the value of `transient-enable-popup-navigation` is `verbose`) when the user navigates to a command using the keyboard.

If OBJ's `summary` slot is a string, that is used. If `summary` is a function, that is called with OBJ as the only argument and the returned string is used. If `summary` is or returns something other than a string or `nil`, no summary is shown. If `summary` is or returns `nil`, the first line of the documentation string is used, if any.

If RETURN is non-`nil`, this function returns the summary instead of showing it. This is used when a tooltip is needed.

5.7 Prefix Slots

Value and Scope

- `default-value` The default value of the prefix. Use the keyword argument `:value` (sic) to set this slot in the definition of a prefix.
- `init-value` A function that is responsible for setting the object's value. If bound, then this is called with the object as the only argument. Usually this is not bound, in which case the object's primary `transient-init-value` method is called instead.
- `history-key` If multiple prefix commands should share a single value, then this slot has to be set to the same value for all of them. You probably don't want that.
- `remember-value` When a suffix command is invoked, which can consume the prefix's value (which depends on the suffix slot `transient` and the prefix slots `transient-suffix` and `transient-non-suffix`), then the value is automatically pushed to the prefix's value history.

This slot allows additionally setting or even saving the value, so that it becomes the initial value when the menu is invoked again.

Beside `nil`, the value can be one of these symbols:

- `export` Set the value when it is exported. That is the time when the value would ordinarily just be pushed to the history stack.
- `exit` Set the value when the menu is exited, except when that is done using a command whose sole purpose is to quit the menu.
- `quit` Set the value when the menu is quit, using a command whose sole purpose is to do so.

The value can also be a list of one or more of these symbols and optionally also the symbol `save`.

- `save` Instead of merely setting the value, save it, so that it will be used in future Emacs sessions. At least one other symbol has to be used together with this.

The value can also be a (quoted) variable, whose value is a list of symbols as described above. Ideally an option should be used, since not all users will find the automatic saving of the value desirable.

- `incompatible` A list of lists. Each sub-list specifies a set of mutually exclusive arguments. Enabling one of these arguments causes the others to be disabled. An argument may appear in multiple sub-lists. Arguments must me given in the same form as used in the `argument` or `argument-format` slot of the respective suffix objects, usually something like `--switch` or `--option=%s`. For options and `transient-switches` suffixes it is also possible to match against a specific value, as returned by `transient-infix-value`, for example, `--option=one`.
- `scope` For some transients it might be necessary to have a sort of secondary value, called a "scope". See `transient-define-prefix`.

Behavior

- `transient-suffix`, `transient-non-suffix` and `transient-switch-frame` play a part when determining whether the currently active transient prefix command remains active/transient when a suffix or arbitrary non-suffix command is invoked. See Section 4.9 [Transient State], page 26.
- `refresh-suffixes` Normally suffix objects and keymaps are only setup once, when the prefix is invoked. Setting this to `t`, causes them to be recreated after every command. This is useful when using `:if...` predicates, and those need to be rerun for some reason. Doing this is somewhat costly, and there is a risk of losing state, so this is disabled by default and still considered experimental.
- `environment` A function used to establish an environment while initializing, refreshing or redisplaying a transient prefix menu. This is useful to establish a cache, in case multiple suffixes require the same expensive work. The provided function is called with at least one argument, the function for which it establishes the environment. It must `funcall` that function with no arguments. During initialization the second argument is the prefix object being initialized. This slot is still experimental.

Appearance

- `display-action` determines how this prefix is displayed, overriding `transient-display-buffer-action`. It should have the same type.
- `mode-line-format` is this prefix's mode line format, overriding `transient-mode-line-format`. It should have the same type.
- `column-widths` is only respected inside `transient-columns` groups and allows aligning columns across separate instances of that. A list of integers.
- `variable-pitch` controls whether alignment is done pixel-wise to account for use of variable-pitch characters, which is useful, e.g., when using emoji.

Documentation

- `show-help`, `man-page` or `info-manual` can be used to specify the documentation for the prefix and its suffixes. The command `transient-help` uses the function `transient-show-help` (which see) to lookup and use these values.
- `suffix-description` can be used to specify a function which provides fallback descriptions for suffixes that lack a description. This is intended to be temporarily used when implementing of a new prefix command, at which time `transient-command-summary-or-name` is a useful value.

Internal

These slots are mostly intended for internal use. They should not be set in calls to `transient-define-prefix`.

- `prototype` When a transient prefix command is invoked, then a clone of that object is stored in the global variable `transient--prefix` and the prototype is stored in the clone's `prototype` slot.
- `command` The command, a symbol. Each transient prefix command consists of a command, which is stored in a symbol's function slot and an object, which is stored in the `transient--prefix` property of the same symbol.
- `level` The level of the prefix commands. The suffix commands whose layer is equal or lower are displayed. See Section 2.7 [Enabling and Disabling Suffixes], page 6.
- `value` The likely outdated value of the prefix. Instead of accessing this slot directly you should use the function `transient-get-value`, which is guaranteed to return the up-to-date value.
- `history` and `history-pos` are used to keep track of historic values. Unless you implement your own `transient-infix-read` method you should not have to deal with these slots.
- `unwind-suffix` is used internally to ensure transient state is properly exited, even in case of an error.

5.8 Suffix Slots

Here we document most of the slots that are only available for suffix objects. Some slots are shared by suffix and group objects, they are documented in Section 5.9 [Predicate Slots], page 41.

Also see Section 5.4 [Suffix Classes], page 31.

Slots of `transient-child`

This is the abstract superclass of `transient-suffix` and `transient-group`. This is where the shared `if*` and `inapt-if*` slots (see Section 5.9 [Predicate Slots], page 41), the `level` slot (see Section 2.7 [Enabling and Disabling Suffixes], page 6), and the `advice` and `advice*` slots (see [Slots of `transient-suffix`], page 39) are defined.

- `parent` The object for the parent group.

Slots of `transient-suffix`

- `key` is the key binding, a string in the format returned by `describe-key` and understood by `kbd`.

  That format is more permissive than the one accepted by `key-valid-p`. Being more permissive makes it possible, for example, to write the key binding, which toggles the `-a` command line argument, as `"-a"`, instead of having to write `"- a"`. Likewise additional spaces can be added, which is not removed when displaying the binding in the menu, which is useful for alignment purposes.
- `command` The command, a symbol.
- `transient` Whether to stay transient. See Section 4.9 [Transient State], page 26.
- `format` The format used to display the suffix in the menu buffer. It must contain the following %-placeholders:
  - `%k` For the key.
  - `%d` For the description.
  - `%v` For the infix value. Non-infix suffixes don't have a value.
- `description` The description, either a string or a function, which is called with zero or one argument (the suffix object), and returns a string.
- `face` Face used for the description. In simple cases it is easier to use this instead of using a function as `description` and adding the styling there. `face` is appended using `add-face-text-property`.
- `show-help` A function used to display help for the suffix. If unspecified, the prefix controls how help is displayed for its suffixes. See also function `transient-show-help`.
- `summary` The summary displayed in the echo area, or as a tooltip. If this is `nil`, which it usually should be, the first line of the documentation string is used instead. See `transient-show-summary` for details.
- `definition` A command, which is used if the body is omitted when defining a command using `transient-define-suffix`.

The following two slots are experimental. They can also be set for a group, in which case they apply to all suffixes in that group, except for suffixes that set the same slot to a non-`nil` value.

- `advice` A function used to advise the command. The advise is called using `(apply advice command args)`, i.e., it behaves like an `"around"` advice.
- `advice*` A function used to advise the command. Unlike `advice`, this advises not only the command body but also its `interactive` spec. If both slots are non-`nil`, `advice` is used for the body and `advice*` is used for the `interactive` form. When advising the

`interactive` spec, called using `(funcall advice #'advice-eval-interactive-spec spec)`.

Slots of `transient-infix`

Some of these slots are only meaningful for some of the subclasses. They are defined here anyway to allow sharing certain methods.

- `argument` The long argument, e.g., `--verbose`.
- `shortarg` The short argument, e.g., `-v`.
- `value` The value. Should not be accessed directly.
- `init-value` Function that is responsible for setting the object's value. If bound, then this is called with the object as the only argument. Usually this is not bound, in which case the object's primary `transient-init-value` method is called instead.
- `unsavable` Whether the value of the suffix is not saved as part of the prefixes.
- `multi-value` For options, whether the option can have multiple values. If this is non-`nil`, then the values are read using `completing-read-multiple` by default and if you specify your own reader, then it should read the values using that function or similar.

  Supported non-`nil` values are:

  - Use `rest` for an option that can have multiple values. This is useful e.g., for an `--` argument that indicates that all remaining arguments are files (such as `git log -- file1 file2`).

    In the list returned by `transient-args` such an option and its values are represented by a single list of the form `(ARGUMENT . VALUES)`.
  - Use `repeat` for an option that can be specified multiple times.

    In the list returned by `transient-args` each instance of the option and its value appears separately in the usual from, for example: `("--another-argument" "--option=first" "--option=second")`.

  In both cases the option's values have to be specified in the default value of a prefix using the same format as returned by `transient-args`, e.g., `("--other" "--o=1" "--o=2" ("--" "f1" "f2"))`.
- `always-read` For options, whether to read a value on every invocation. If this is `nil`, then options that have a value are simply unset and have to be invoked a second time to set a new value.
- `allow-empty` For options, whether the empty string is a valid value.
- `history-key` The key used to store the history. This defaults to the command name. This is useful when multiple infixes should share the same history because their values are of the same kind.
- `reader` The function used to read the value of an infix. Not used for switches. The function takes three arguments, *PROMPT*, *INITIAL-INPUT* and *HISTORY*, and must return a string.
- `prompt` The prompt used when reading the value, either a string or a function that takes the object as the only argument and which returns a prompt string.
- `choices` A list of valid values, or a function that returns such a list. The latter is not implemented for `transient-switches`, because I couldn't think of a use-case. How exactly the choices are used varies depending on the class of the suffix.

Slots of `transient-variable`

- `variable` The variable.

Slots of `transient-switches`

- `argument-format` The display format. Must contain `%s`, one of the `choices` is substituted for that. E.g., `--%s-order`.
- `argument-regexp` The regexp used to match any one of the switches. E.g., `\\(--\\(topo\\|author-date\\|date\\)-order\\)`.

5.9 Predicate Slots

Suffix and group objects share two sets of predicate slots that control whether a group or suffix should be available depending on some state. Only one slot from each set can be used at the same time. It is undefined which slot is honored if you use more than one.

Predicates from the first group control whether the suffix is present in the menu at all.

- `if` Enable if predicate returns non-`nil`.
- `if-not` Enable if predicate returns `nil`.
- `if-non-nil` Enable if variable's value is non-`nil`.
- `if-nil` Enable if variable's value is `nil`.
- `if-mode` Enable if major-mode matches value.
- `if-not-mode` Enable if major-mode does not match value.
- `if-derived` Enable if major-mode derives from value.
- `if-not-derived` Enable if major-mode does not derive from value.

Predicates from the second group control whether the suffix can be invoked. The suffix is shown in the menu regardless, but when it is considered `"inapt"`, then it is grayed out to indicated that it currently cannot be invoked.

- `inapt-if` Inapt if predicate returns non-`nil`.
- `inapt-if-not` Inapt if predicate returns `nil`.
- `inapt-if-non-nil` Inapt if variable's value is non-`nil`.
- `inapt-if-nil` Inapt if variable's value is `nil`.
- `inapt-if-mode` Inapt if major-mode matches value.
- `inapt-if-not-mode` Inapt if major-mode does not match value.
- `inapt-if-derived` Inapt if major-mode derives from value.
- `inapt-if-not-derived` Inapt if major-mode does not derive from value.

By default these predicates run when the prefix command is invoked, but this can be changes, using the `refresh-suffixes` prefix slot. See Section 5.7 [Prefix Slots], page 36.

One more slot is shared between group and suffix classes, `level`. Like the slots documented above, it is a predicate, but it is used for a different purpose. The value has to be an integer between 1 and 7. `level` controls whether a suffix or a group should be available depending on user preference. See Section 2.7 [Enabling and Disabling Suffixes], page 6.

Appendix A FAQ

A.1 Can I control how the menu buffer is displayed?

Yes, see `transient-display-buffer-action` in Section 2.9 [Configuration], page 9. You can also control how the menu buffer is displayed on a case-by-case basis by passing `:display-action` to `transient-define-prefix`.

A.2 How can I copy text from the menu buffer?

To be able to mark text in Transient's menu buffer using the mouse, you have to add the below binding. Note that for technical reasons, the region won't be visualized, while doing so. After you have quit the transient menu, you will be able to yank it in another buffer.

```
(keymap-set transient-predicate-map
            "<mouse-set-region>"
            #'transient--do-stay)
```

Copying the region while not seeing the region is a bit fiddly, so a dedicated command, `transient-copy-menu-text`, was added. You have to add a binding for this command in `transient-map`.

```
(keymap-set transient-map "C-c C-w" #'transient-copy-menu-text)
```

A.3 How can I autoload prefix and suffix commands?

If your package only supports Emacs 30, just prefix the definition with `;;;###autoload`. If your package supports released versions of Emacs, you unfortunately have to use a long form autoload comment as described in Section "Autoload" in `elisp`.

```
;;;###autoload (autoload 'magit-dispatch "magit" nil t)
(transient-define-prefix magit-dispatch ()
  ...)
```

A.4 How does Transient compare to prefix keys and universal arguments?

See `https://github.com/magit/transient/wiki/Comparison-with-prefix-keys-and-universal-argume`

A.5 How does Transient compare to Magit-Popup and Hydra?

See `https://github.com/magit/transient/wiki/Comparison-with-other-packages`.

A.6 Why does `q` not quit popups anymore?

I agree that `q` is a good binding for commands that quit something. This includes quitting whatever transient is currently active, but it also includes quitting whatever it is that some specific transient is controlling. The transient `magit-blame` for example binds `q` to the command that turns `magit-blame-mode` off.

So I had to decide if `q` should quit the active transient (like Magit-Popup used to) or whether `C-g` should do that instead, so that `q` could be bound in individual transient to whatever commands make sense for them. Because all other letters are already reserved for use by individual transients, I have decided to no longer make an exception for `q`.

If you want to get `q`'s old binding back then you can do so. Doing that is a bit more complicated than changing a single key binding, so I have implemented a function, `transient-bind-q-to-quit` that makes the necessary changes. See its documentation string for more information.

Appendix B Keystroke Index

Appendix C Command and Function Index

Appendix D Variable Index

Appendix E Concept Index

Appendix F GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see https://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.