Transient User and Developer Manual

for version 0.1.0 (v0.1.0-39-gb1da0ca+1)

Jonas Bernoulli

Table of Contents

1 Introduction

Taking inspiration from prefix keys and prefix arguments, Transient implements a similar abstraction involving a prefix command, infix arguments and suffix commands. We could call this abstraction a "transient command", but because it always involves at least two commands (a prefix and a suffix) we prefer to call it just a "transient".

> Transient keymaps are a feature provided by Emacs. Transients as implemented by this package involve the use of transient keymaps.
>
> Emacs provides a feature that it calls "prefix commands". When we talk about "prefix commands" in this manual, then we mean our own kind of "prefix commands", unless specified otherwise. To avoid ambiguity we sometimes use the terms "transient prefix command" for our kind and "regular prefix command" for Emacs' kind.

When the user calls a transient prefix command, then 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 popup 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 popup 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 it can choose to use or ignore the prefix arguments. For a suffix command that was invoked from a transient the variable `current-transient-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`.

The information shown in the popup buffer while a transient is active looks a bit like this:

```
,----------------------------------------
|Arguments
| -f Force (--force)
| -a Annotate (--annotate)
|
|Create
| t tag
| r telease
`----------------------------------------
```

> This is a simplified version of `magit-tag`. Info manuals do not support images or colored text, so the above "screenshot" lacks some information; in practice you would be able to tell whether the arguments `--force` and `--annotate` are enabled or not based on their color.

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 popup buffer. 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` 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, then one can call a command with arguments that are just as complex as when calling the same function non-interactively using code.

Invoking a transient 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 5) 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).

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 commands activates a transient keymap, which temporarily binds the transient's infix and suffix commands. 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 a popup 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.

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 popup 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, then 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 doc-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 (`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 in the popup buffer by default.

Most of these commands are bound to `C-x <key>` and after pressing `C-x` a section featuring all common commands is temporarily shown in the popup buffer. After invoking one of these commands, that section disappears again. Note however that one of these commands is described as "Show common permanently"; invoke that if you want the common commands to always be shown for all transients.

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

`transient-show-common-commands` [User Option]
This option controls whether shared suffix commands are shown alongside the transient-specific infix and suffix commands. By default the shared commands are not shown to avoid overwhelming the user with to many options.

While a transient is active, pressing `C-x` always shows the common command. The value of this option can be changed for the current Emacs session by typing `C-x t` while a transient is active.

The other common commands are describe in either the previous node or in one of the following nodes.

Some of Transient's key bindings differ from the respective bindings of Magit-Popup; see Appendix A [FAQ], page 31, for more information.

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]

[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. This is something I am rethinking, but I don't want to rush any changes.)

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).

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

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

`transient-values-file` [User Option]
: This file 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. This values can be cycled through the same way one can cycle through the history of commands that read user-input in the minibuffer.

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

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

In addition to the transient-wide history, Transient of course supports per-infix history. When an infix reads user-input using the minibuffer, then the user can use the regular minibuffer 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 6.7 [Suffix Slots], page 23).

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

`transient-history-file` [User Option]
: This file 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 in 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, then typing `<key>` shows information about the suffix command that `<key>` normally is bound to (instead of invoking it). Pressing `C-h` a second time shows information about the *prefix* command.

After typing `<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 commands doc-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 transient and then pressing `C-x l` to enter the "edit" mode, see 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.

(Magit currently places nearly all suffixes on level 4 and lower levels are not used at all yet. So for the time being you should not set a lower default level and using a higher level might not give you as many additional suffixes as you hoped.)

`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 file is used to persist the levels of transients and their suffixes between Emacs sessions.

`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 does instead prompt the user for the level that 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.

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 popup 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 popup window down ARG lines. If ARG is `nil`, then it scrolls near full screen. This is a wrapper around `scroll-down-command` (which see).

3 Other Options

`transient-show-popup` [User Option]

This option controls whether the current transient's infix and suffix commands are shown in the popup buffer.

If `t` (the default), then the infix and suffix commands are shown as soon as the transient is invoked. If `nil`, only a one line summary is shown until the user presses a key that forms an incomplete key sequence. If a number, behave as for `nil` but also show the commands after that many seconds of inactivity.

`transient-display-buffer-action` [User Option]

This option specifies the action used to display the transient popup buffer. The transient popup buffer is displayed in a window using `(display-buffer buf 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`.

The default is `(display-buffer-in-side-window (side . bottom))`. This displays the window at the bottom of the selected frame. Another useful value is `(display-buffer-below-selected)`. This is what `magit-popup` used by default. For more alternatives see Section "Display Action Functions" in `elisp`.

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 it doesn't work.

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

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

This option controls whether the transient popup 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), then the buffer also has no mode-line, but a thin line is drawn instead, using the background color of the face `transient-separator`.

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

`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.

When this option is non-nil, then 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. It 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 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-detect-key-conflicts` [User Option]

This option controls whether key binding conflicts should be detected at the time the transient is invoked. If so, then 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.

4 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.

The following functions share a few arguments:

- PREFIX is a transient prefix command, a symbol.
- SUFFIX is a transient infix or suffix specification in the same form as expected by `define-transient-command`. 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 5.2.2 [Suffix Specifications], page 13.
- LOC is a command, a key vector or a key description (a string as returned by `key-description`).

These functions operate on the information stored in the `transient--layout` property of the PREFIX symbol. Suffix entries in that tree are not objects but have the form `(LEVEL CLASS PLIST)`, where plist should set at least `:key`, `:description` and `:command`.

`transient-insert-suffix` *prefix loc suffix* [Function]
: This function inserts SUFFIX into PREFIX before LOC.

`transient-append-suffix` *prefix loc suffix* [Function]
: This function inserts SUFFIX into PREFIX after LOC.

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

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

`transient-get-suffix` *prefix loc* [Function]
: This function returns the suffix 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 at LOC in PREFIX, by setting the PROP of its plist to VALUE.

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` may signal an error.

5 Defining New Commands

5.1 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** it 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.

`define-transient-command` *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.

GROUPs add key bindings for infix and suffix commands and specify how these bindings are presented in the popup buffer. At least one GROUP has to be specified. See Section 5.2 [Binding Suffix and Infix Commands], page 11.

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.

5.2 Binding Suffix and Infix Commands

The macro `define-transient-command` is used to define a transient. This defines the actual transient prefix command (see Section 5.1 [Defining Transients], page 11) 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 4 [Modifying Existing Transients], page 10). These functions take a "suffix specification" as one of their arguments, which has the same form as the specifications used in `define-transient-command`.

5.2.1 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 6.1 [Group Classes], page 19.

Groups are specified in the call to `define-transient-command`, 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...` 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 6.8 [Predicate Slots], page 24.
- Finally 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 ELEMENTs are either all subgroups (vectors), or all suffixes (lists) 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 ELEMENTs are not subgroups, then they can be a mixture of lists that specify commands and strings. Strings are inserted verbatim. The empty string can be used to insert gaps between suffixes, which is particularly useful if the suffixes are outlined as a table.

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

5.2.2 Suffix Specifications

A transient's suffix and infix commands are bound when the transient prefix command is defined using `define-transient-command`, see Section 5.1 [Defining Transients], page 11. The commands are organized into groups, see Section 5.2.1 [Group Specifications], page 12. 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 4 [Modifying Existing Transients], page 10.

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 KEYWORDs `: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 objects value 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, either a vector or key description string.
- DESCRIPTION is the description, either a string or a function that 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 is a symbol that is bound as a function, which has to be a command. Any command will do; it does not need to have an object associated with it (as would be the case if `define-suffix-command` or `define-infix-command` were used to define it).

  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.

- The mandatory argument can also be an 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 the long argument (which can also be specified using `:argument`).

Only the long argument is displayed in the popup 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 6.7 [Suffix Slots], page 23.

5.3 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".

`define-suffix-command` *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. Use the function `transient-args` or the low-level variable `current-transient-suffixes` if the former does not give you all the required details. This should, but does not necessarily have to be, done inside the `interactive` form; just like for `prefix-arg` and `current-prefix-arg`.

`define-infix-command` *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.

The keyword-value pairs are mandatory. All transient infix commands are `equal` to each other (but not `eq`), so it is meaningless to define an infix command without also setting at least `:class` and one other keyword (which it is depends on the used class, usually `:argument` or `:variable`).

Each keyword has to be a keyword symbol, either `:class` or a keyword argument supported by the constructor of that class. The `transient-switch` class is used if the class is not specified explicitly.

The function definitions 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-suffix-command` to define the infix command and use `t` as the value of the `:transient` keyword.

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

This macro defines NAME as a transient infix command.

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

5.4 Using Infix Arguments

The function 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' 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` **&optional** *prefix separate* [Function]

This function returns the value of the transient from which the current suffix was called. If the current suffix command was not called from a transient, then it returns `nil`.

If optional PREFIX is non-`nil`, then it should be a symbol, a transient prefix command. In that case the value of the transient is only returned if the suffix was invoked from **that** transient. Otherwise `nil` is returned. This function is also used internally, in which PREFIX can also be a `transient-prefix` object.

If optional SEPARATE is non-`nil`, then the arguments are separated into two groups. If SEPARATE is `t`, they are separated into atoms and conses (`nil` isn't a valid value, so it doesn't matter that that is both an atom and a cons).

SEPARATE can also be a predicate function, in which case the first element is a list of the values for which it returns non-`nil` and the second element is a list of the values for which it returns `nil`.

For transients that are used to pass arguments to a subprocess (such as `git`), `stringp` is a useful value for SEPARATE, it separates non-positional arguments from positional arguments. The value of Magit's file argument (`"--"`) for example looks like this: `("--" file...)`."

`current-transient-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.

`current-transient-prefix` [Variable]
: The transient from which this suffix command was invoked. The returned value is a `transient-prefix` object, which holds information associated with the transient prefix command.

`current-transient-command` [Variable]
: The transient from which this suffix command was invoked. The returned value is a symbol, the transient prefix command.

5.5 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.

But these are just the defaults. Whether a certain command deactivates or "exits" the transient is configurable. There is more than one way in which a command can be "transient" or "non-transient"; the exact behavior is implemented by calling a so-called "pre-command" function. Whether non-suffix commands are allowed to be called is configurable per transient.

- The transient-ness of suffix commands (including infix commands) is controlled by the value of their `transient` slot, which can be set either when defining the command or when adding a binding to a transient while defining the respective transient prefix command.

  Valid values are booleans and the pre-commands described below.

  - `t` is equivalent to `transient--do-stay`.
  - `nil` is equivalent to `transient--do-exit`.
  - If `transient` is unbound (and that is actually the default for non-infix suffixes) then the value of the prefix's `transient-suffix` slot is used instead. The default value of that slot is `nil`, so the suffix's `transient` slot being unbound is essentially equivalent to it being `nil`.
- A suffix command can be a prefix command itself, i.e. a "sub-prefix". While a sub-prefix is active we nearly always want `C-g` to take the user back to the "super-prefix".

However in rare cases this may not be desirable, and that makes the following complication necessary:

For `transient-suffix` objects the `transient` slot is unbound. We can ignore that for the most part because, as stated above, `nil` and the slot being unbound are equivalent, and means "do exit". That isn't actually true for suffixes that are sub-prefixes though. For such suffixes unbound means "do exit but allow going back", which is the default, while `nil` means "do exit permanently", which requires that slot to be explicitly set to that value.

- 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 `transient` slot.

The available pre-command functions are documented below. 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`.

5.5.1 Pre-commands for Infixes

The default for infixes is `transient--do-stay`. This is also the only function that makes sense for infixes.

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

5.5.2 Pre-commands for Suffixes

The default for suffixes is `transient--do-exit`.

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

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

`transient--do-replace` [Function]
Call the transient prefix command, replacing the active transient.

This is used for suffix that are prefixes themselves, i.e. for sub-prefixes.

5.5.3 Pre-commands for Non-Suffixes

The default for non-suffixes, i.e commands that are bound in other keymaps beside the transient keymap, is `transient--do-warn`. Silently ignoring the user-error is also an option, though probably not a good one.

If you want to let the user invoke non-suffix commands, then use `transient--do-stay` as the value of the prefix's `transient-non-suffix` slot.

`transient--do-warn` [Function]
Call `transient-undefined` and stay transient.

`transient--do-noop` [Function]
Call `transient-noop` and stay transient.

5.5.4 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`.

6 Classes and Methods

Transient uses classes and generic functions to make it possible to define new types of suffix commands that are similar to existing types, but behave differently in some aspects. It does the same for groups and prefix commands, though at least for prefix commands that appears to be less important.

Every prefix, infix and suffix command is associated with an object, which holds information that 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.

6.1 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 `define-transient-command`.

- 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 (aka "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.

6.2 Group Methods

`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 popup 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 temporally make the `transient--source-buffer` the current buffer.

6.3 Prefix Classes

Currently the `transient-prefix` class is being used for all prefix command and there is only a single generic functions that can be specialized based on the class of a prefix command.

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

This generic function is called while setting up the transient and is responsible for initializing the `history` slot. This is the transient-wide history; many individual infixes also have a history of their own.

The default (and currently only) method extracts the value from the global variable `transient-history`.

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.

6.4 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 6.1 [Group Classes], page 19).
- All infix classes derived 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 5.5 [Transient State], page 16). 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 5.3 [Defining Suffix and Infix Commands], page 14).

- 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-variables` class.

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.

6.5 Suffix Methods

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

6.5.1 Suffix Value Methods

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

This generic function sets the initial value of the object OBJ.

This function is called for all suffix commands, but unless a concrete method is implemented this falls through to the default implementation, which is a noop. In other words this usually only does something for infix commands, but note that this is not implemented for the abstract class `transient-infix`, so if your class derives from that directly, then you must implement a method.

`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 suffix object OBJ.

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 suffix commands, but unless a concrete method is implemented this falls through to the default implementation, which is a noop.

6.5.2 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.

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 doc-string.

For suffixes show the command's doc-string.

For infixes show the manpage if that is specified. Otherwise show the command's doc-string.

6.6 TODO Prefix Slots

6.7 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 6.8 [Predicate Slots], page 24.

Also see Section 6.4 [Suffix Classes], page 20.

6.7.1 Slots of `transient-suffix`

- `key` The key, a key vector or a key description string.
- `command` The command, a symbol.
- `transient` Whether to stay transient. See Section 5.5 [Transient State], page 16.
- `format` The format used to display the suffix in the popup 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 that is called with no argument and returns a string.

6.7.2 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`.
- `multi-value` For options, whether the option can have multiple values. If non-nil, then default to use `completing-read-multiple`.
- `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. How exactly that is used depends on the class of the object.

6.7.3 Slots of `transient-variable`

- `variable` The variable.

6.7.4 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\\)`.

6.8 Predicate Slots

Suffix and group objects share some predicate slots that control whether a group or suffix should be available depending on some state. Only one of these slots can be used at the same time. It is undefined what happens if you use more than one.

- `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.

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 it should be available depending on whether the user wants that or not. See Section 2.7 [Enabling and Disabling Suffixes], page 6.

7 Related Abstractions and Packages

7.1 Comparison With Prefix Keys and Prefix Arguments

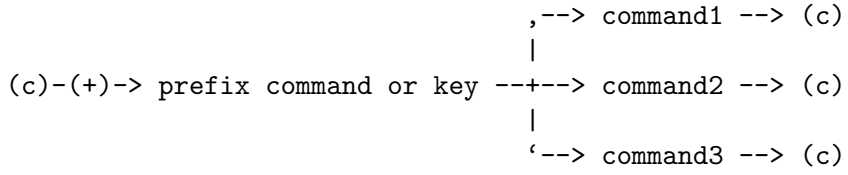
While transient commands were inspired by regular prefix keys and prefix arguments, they are also quite different and much more complex.

The following diagrams illustrate some of the differences.

- `(c)` represents a return to the command loop.
- `(+)` represents the user's choice to press one key or another.
- `{WORD}` are possible behaviors.
- `{NUMBER}` is a footnote.

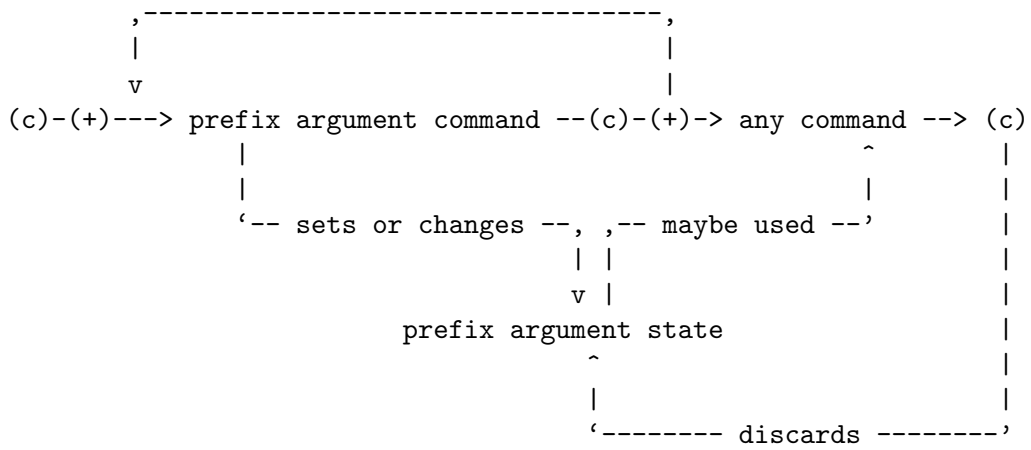
7.1.1 Regular Prefix Commands

See Section "Prefix Keys" in `elisp`.



7.1.2 Regular Prefix Arguments

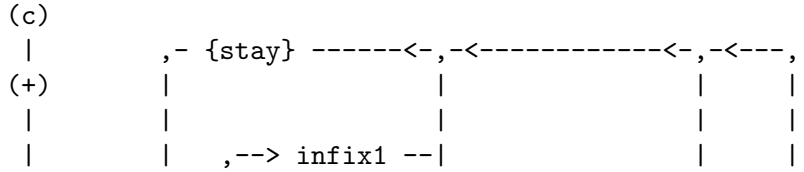
See Section "Prefix Command Arguments" in `elisp`.

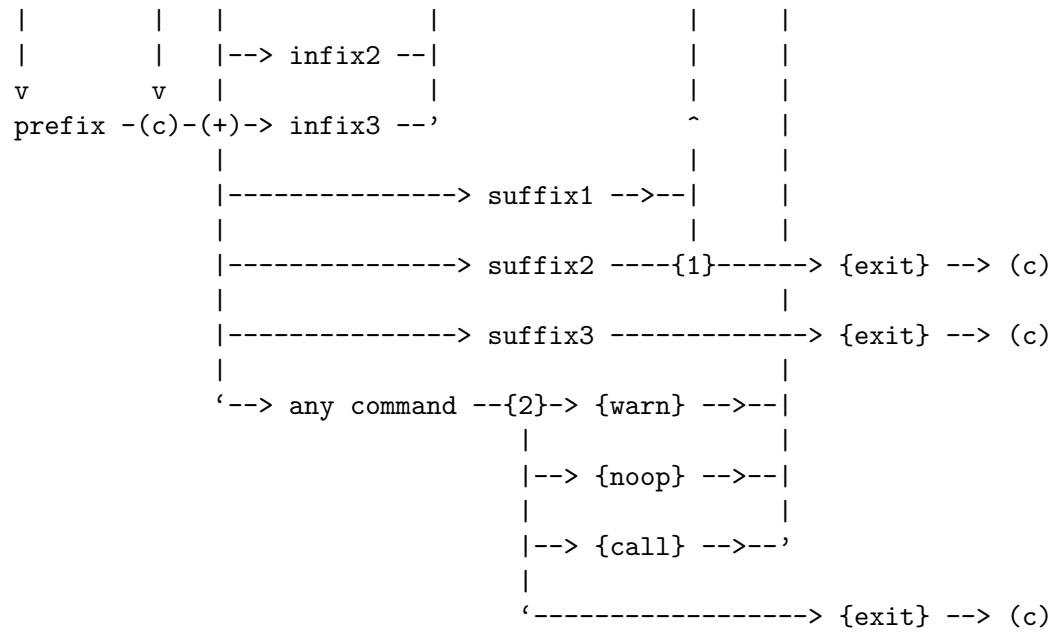


7.1.3 Transients

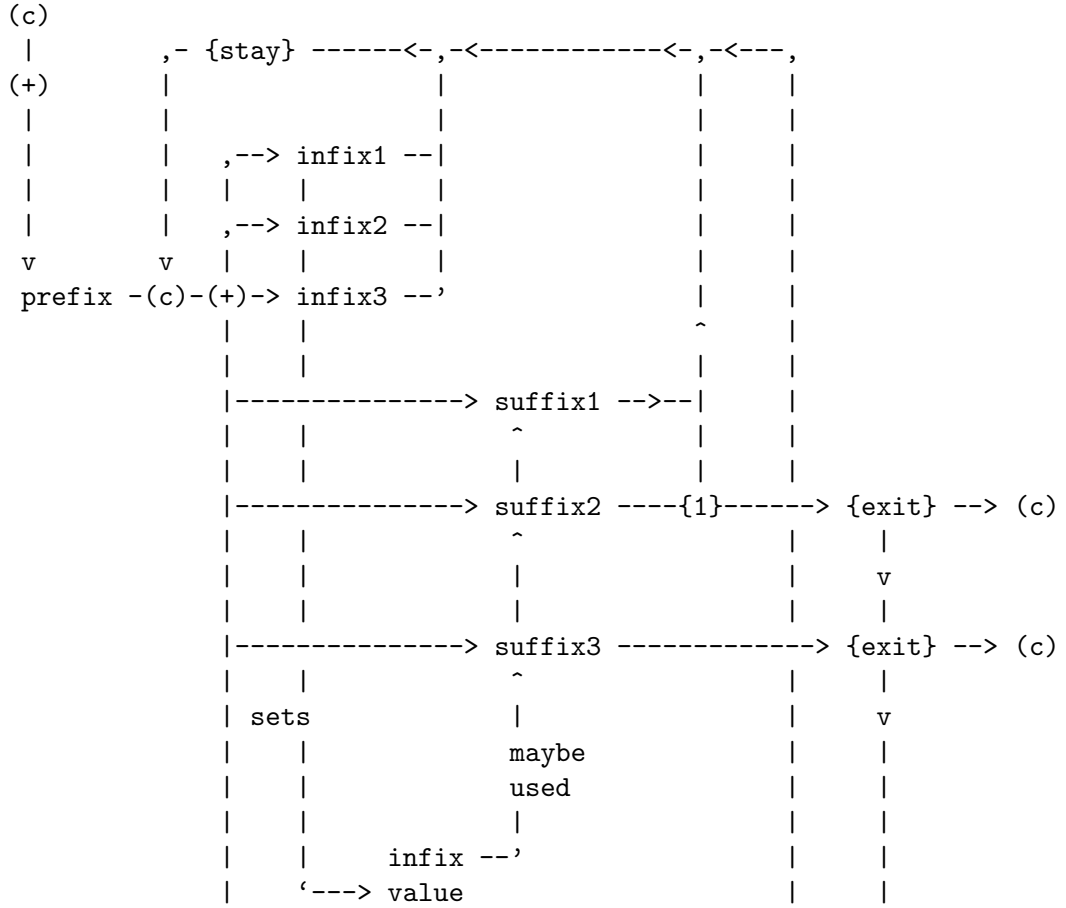
(∩-´)⊃.*

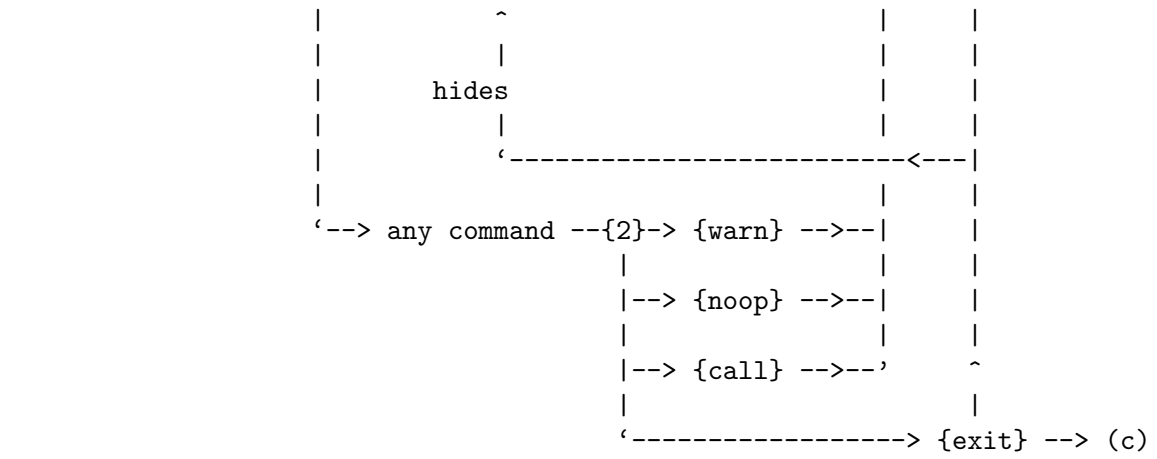
This diagram ignores the infix value and external state:



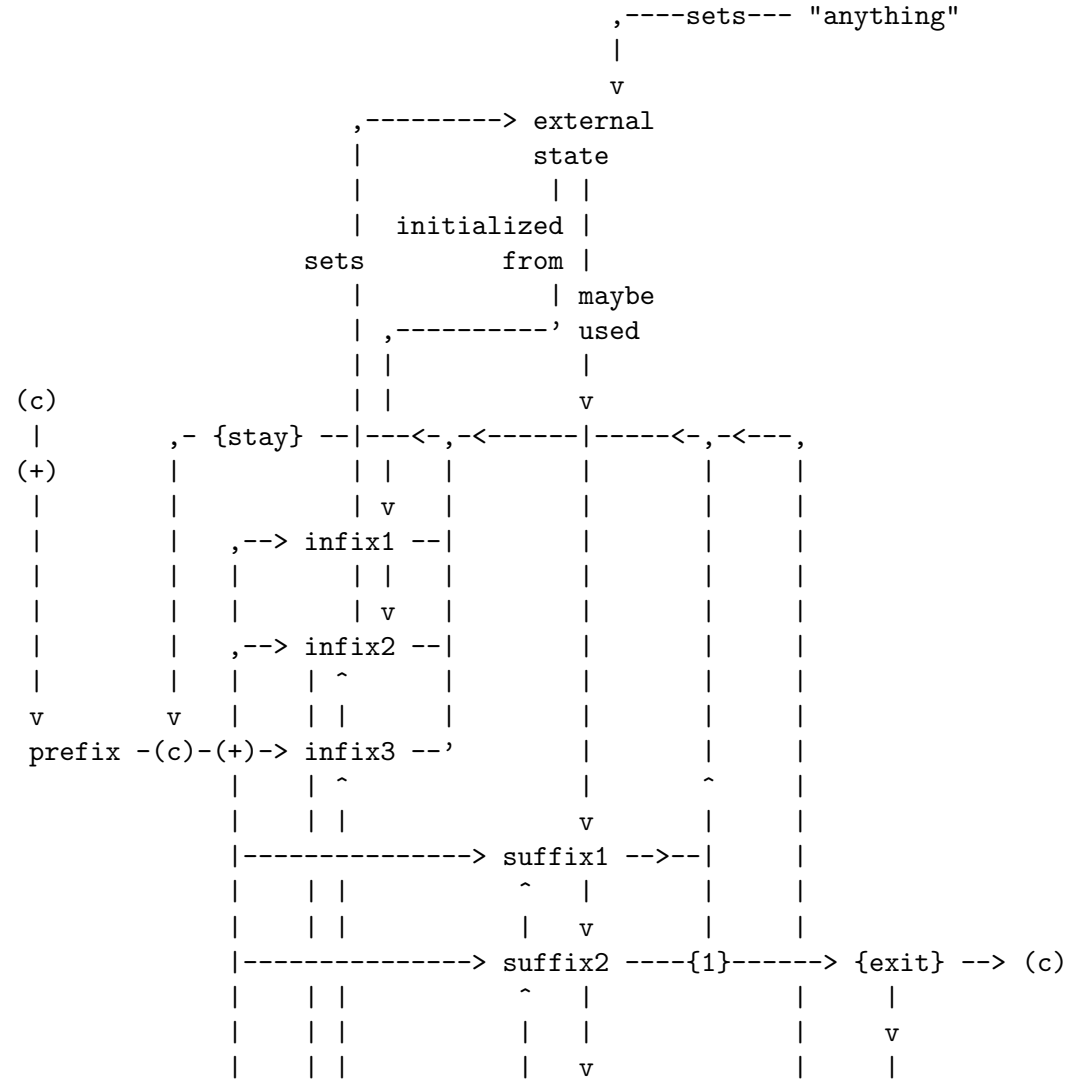


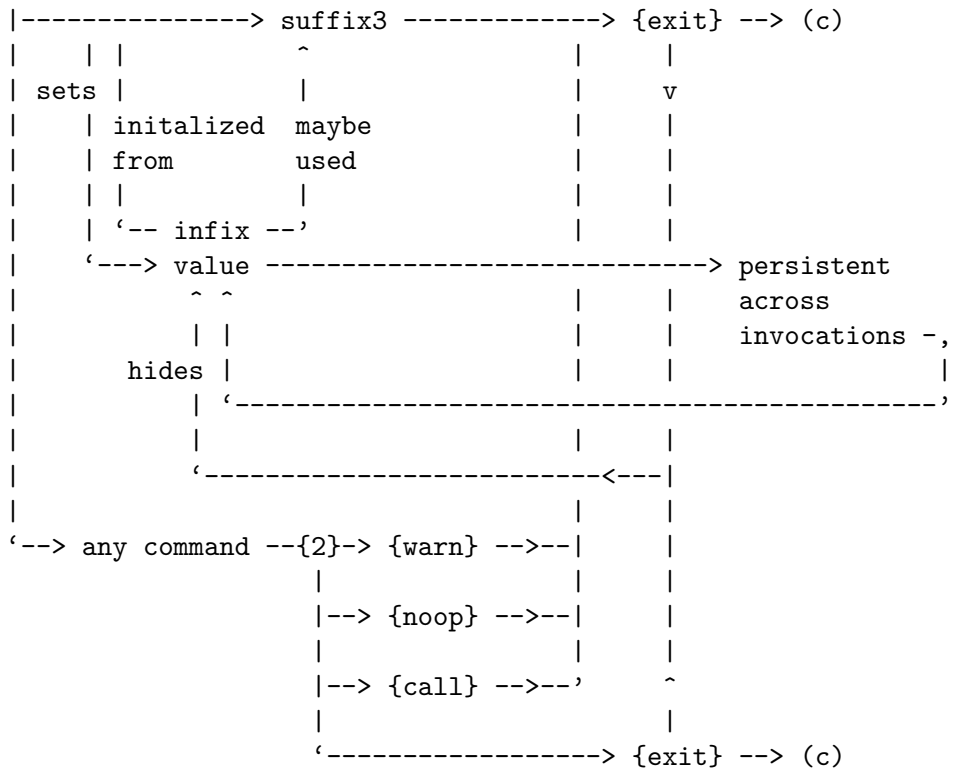
This diagram takes the infix value into account to an extend, while still ignoring external state:





This diagram provides more information about the infix value and also takes external state into account.





- {1} Transients can be configured to be exited when a suffix command is invoked. The default is to do so for all suffixes except for those that are common to all transients and which are used to perform tasks such as providing help and saving the value of the infix arguments for future invocations. The behavior can also be specified for individual suffix commands and may even depend on state.
- {2} Transients can be configured to allow the user to invoke non-suffix commands. The default is to not allow that and instead warn the user.

Despite already being rather complex, even the last diagram leaves out many details. Most importantly it implies that the decision whether to remain transient is made later than it actually is made (for the most part a function on `pre-command-hook` is responsible). But such implementation details are of little relevance to users and are covered elsewhere.

7.2 Comparison With Other Packages

7.2.1 Magit-Popup

Transient is the successor to Magit-Popup (see `magit-popup`).

One major difference between these two implementations of the same ideas is that while Transient uses transient keymaps and embraces the command-loop, Magit-Popup implemented an inferior mechanism that does not use transient keymaps and that instead of using the command-loop implements a naive alternative based on `read-char`.

Magit-Popup does not use classes and generic functions and defining a new command type is near impossible as it involves adding hard-coded special-cases to many functions.

Because of that only a single new type was added, which was not already part of Magit-Popup's initial release.

A lot of things are hard-coded in Magit-Popup. One random example is that the key bindings for switches must begin with `"-"` and those for options must begin with `"="`.

7.2.2 Hydra

Hydra (see `https://github.com/abo-abo/hydra`) is another package that provides features similar to those of Transient.

Both packages use transient keymaps to make a set of commands temporarily available and show the available commands in a popup buffer.

A Hydra "body" is equivalent to a Transient "prefix" and a Hydra "head" is equivalent to a Transient "suffix". Hydra has no equivalent of a Transient "infix".

Both hydras and transients can be used as simple command dispatchers. Used like this they are similar to regular prefix commands and prefix keys, except that the available commands are shown in the popup buffer.

(Another package that does this is `which-key`. It does so automatically for any incomplete key sequence. The advantage of that approach is that no additional work is necessary; the disadvantage is that the available commands are not organized semantically.)

Both Hydra and Transient provide features that go beyond simple command dispatchers:

- Invoking a command from a hydra does not necessarily exit the hydra. That makes it possible to invoke the same command again, but using a shorter key sequence (i.e. the key that was used to enter the hydra does not have to be pressed again).

  Transient supports that too, but for now this feature is not a focus and the interface is a bit more complicated. A very basic example using the current interface:

  ```
  (define-transient-command outline-navigate ()
    :transient-suffix     'transient--do-stay
    :transient-non-suffix 'transient--do-warn
    [("p" "next visible heading" outline-previous-visible-heading)
     ("n" "next visible heading" outline-next-visible-heading)])
  ```

- Transient supports infix arguments; values that are set by infix commands and then consumed by the invoked suffix command(s).

  To my knowledge, Hydra does not support that.

Both packages make it possible to specify how exactly the available commands are outlined:

- With Hydra this is often done using an explicit format string, which gives authors a lot of flexibility and makes it possible to do fancy things.

  The downside of this is that it becomes harder for a user to add additional commands to an existing hydra and to change key bindings.

- Transient allows the author of a transient to organize the commands into groups and the use of generic functions allows authors of transients to control exactly how a certain command type is displayed.

  However while Transient supports giving sections a heading it does not currently support giving the displayed information more structure by, for example, using box-drawing characters.

That could be implemented by defining a new group class, which lets the author specify a format string. It should be possible to implement that without modifying any existing code, but it does not currently exist.

Appendix A FAQ

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

Yes, see `transient-display-buffer-action` in Chapter 3 [Other Options], page 8.

A.2 Why did some of the key bindings change?

You may have noticed that the bindings for some of the common commands do **not** have the prefix `C-x` and that furthermore some of these commands are grayed out while others are not. That unfortunately is a bit confusing if the section of common commands is not shown permanently, making the following explanation necessary.

The purpose of usually hiding that section but showing it after the user pressed the respective prefix key is to conserve space and not overwhelm users with too much noise, while allowing the user to quickly list common bindings on demand.

That however should not keep us from using the best possible key bindings. The bindings that do use a prefix do so to avoid wasting too many non-prefix bindings, keeping them available for use in individual transients. The bindings that do not use a prefix and that are **not** grayed out are very important bindings that are **always** available, even when invoking the "common command key prefix" or **any other** transient-specific prefix. The non-prefix keys that **are** grayed out however, are not available when any incomplete prefix key sequence is active. They do not use the "common command key prefix" because it is likely that users want to invoke them several times in a row and e.g. `M-p M-p M-p` is much more convenient than `C-x M-p C-x M-p C-x M-p`.

You may also have noticed that the "Set" command is bound to `C-x s`, while Magit-Popup used to bind `C-c C-c` instead. I have seen several users praise the latter binding (sic), so I did not change it willy-nilly. The reason that I changed it is that using different prefix keys for different common commands, would have made the temporary display of the common commands even more confusing, i.e. after pressing `C-c` all the `C-x ...` bindings would be grayed out.

Using a single prefix for common commands key means that all other potential prefix keys can be used for transient-specific commands **without** the section of common commands also popping up. `C-c` in particular is a prefix that I want (and already do) use for Magit, and also using that for a common command would prevent me from doing so.

(Also see the next question.)

A.3 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 doc-string for more information.

Appendix B Keystroke Index

C

M

Appendix C Command Index

Appendix D Function Index

D

T

Appendix E Variable Index

C

T