The Stump Window Manager

Shawn Betts, David Bjergaard

1 Introduction

StumpWM is a manual, tiling X11 window manager written entirely in Common Lisp. Unlike traditional window managers, StumpWM places windows in order to maximize the amount of the screen used. The window layouts managed by StumpWM are defined by the user in much the same way that windows are managed by GNU screen, or emacs.

Before StumpWM, there was ratpoison, another tiling window manager written entirely in C. StumpWM grew out of the authors' frustration with writing ratpoison in C. Very quickly we realized we were building into ratpoison lispy-emacs style paradigms. StumpWM's goals are similar to ratpoison's but with an emphasis on customizability, completeness, and cushiness.

1.1 Starting StumpWM

There are a number of ways to start StumpWM but the most straight forward method is as follows. This assumes you have a copy of the StumpWM source code and are using the '`SBCL`' Common Lisp environment.

1. Install the prerequisites and build StumpWM as described in `README`. This should give you a `stumpwm` executable.
2. In your `~/.xinitrc` file include the line `/path/to/stumpwm`. Remember to replace '`/path/to/`' with the actual path.
3. Finally, start X windows with `startx`. Cross your fingers. You should see a '`Welcome To the Stump Window Manager`' message pop up in the upper, right corner. At this point, you have successfully started StumpWM.

`stumpwm` **&optional** (*display-str* **(or (getenv DISPLAY) :0)**) [Function]

Start the stump window manager.

1.2 Basic Usage

Once you have StumpWM up and running, the first thing you might want to do is start `emacs`. Type `C-t e`, or in other words `Control + t` followed by `e`. Now perhaps you want an `xterm`. Type `C-t c`. Now you have some programs running.

To see a list of windows StumpWM is managing, type `C-t w`. The highlighted window is the one you're looking at right now. It's the focused window.

All of StumpWM's keys are bound to named commands, which can be executed not only by keys but also from the input bar. Type `C-t ;` to open a command prompt. Now type `time` and press return. Note, `time` can also be called by typing `C-t a`. Throughout this manual you'll find definitions for commands, functions, and variables. Any command you see in this manual can be executed from the input bar or bound to a key.

At this point you probably want to switch back from your new `xterm` to `emacs`. Type `C-t C-t`. This runs the `other` command. Type it again and you're back to xterm.

Perhaps you'd like to see `emacs` and `xterm` side-by-side. Type `C-t s`. You have now split the screen into 2 `frames`. For more information see Chapter 6 [Frames], page 45. To switch to the empty frame type `C-t TAB`. Now let's pull the xterm window into this empty frame.

Type `C-t w` for a window listing. Find the `xterm` window. See the number beside it? Type `C-t` followed by `xterm`'s window number.

Another common activity is browsing the internet. Type `C-t !`. The input bar pops up again. You can now run a shell command. Let's start a web browser: type `firefox` into the input bar and press return.

Unfortunately, `firefox` probably isn't wide enough because it's in one of the frames. Type `C-t Q` to remove all frames but the current one and resize it to fit the screen.

For a full list of key bindings, see Section 2.1 [List of Default Keybindings], page 9.

1.3 Basic Concepts

An introduction to some of the basic concepts used by StumpWM.

1.3.1 Screens and Heads

A screen is an Xlib concept representing a section of video memory onto which physical monitors, called "heads", are mapped. A screen can be thought of as an abstract rectangle containing all the heads arranged in a particular layout.

With most modern systems, you'll only have a single screen no matter how many heads are connected to your computer. Each head will have its own frame, and you can move between heads using the normal frame movement commands.

The layout of the heads within the screen can be specified in one of two ways: either at startup using your system's Xorg configuration files, or on the fly using tools like XRandR. If the computer is booted with multiple monitors attached, but without specifying a layout for them, they will all show identical output.

StumpWM will attempt to detect the layout of the heads once at startup, or any time a RandR command is issued.

In rarer setups you may have multiple screens, with one head per screen. That means that you'll move between heads using screen movement commands ('snext', 'sprev', and 'sother') rather than frame movement commands.

1.3.2 Group Basics

A group is usually referred to as a "desktop" or "workspace" in other window managers. StumpWM starts with a single group, called "Default". Each group has its own configuration of frames and windows that is separate from and independent of other groups. You can't have different groups display in different monitors: when you switch groups, all monitors switch to that group.

Each group contains an ordered list of frames.

1.3.3 Floating Group Basics

Within a floating group, windows behave more like they do in traditional window managers: rather than being arranged into frames, they each have their own box, which can be freely resized and repositioned, and allowed to overlap. Each window has a thicker border at the top. Left click in this border and drag to move the window, or right click and drag to resize it.

A modifier key can be used to perform the move and resize operations by clicking in the window itself instead of on its top border. The default modifier is super, and can be configured with **float-window-modifier**.

`*float-window-modifier*` [Variable]

The keyboard modifier to use for resize and move floating windows without clicking on the top border. Valid values are :META :ALT :HYPER :SUPER, :ALTGR and :NUMLOCK.

Most of the window-switching commands listed below do not function in a floating group. You're restricted to 'other', the 'select-window-*' commands, and 'windowlist'.

1.3.4 Dynamic Group Basics

Within a dynamic group, windows are organized into a *master window* and a *window stack*, with one of each per head. When a new window is added to a head within a dynamic group, that heads master window is pushed onto that heads window stack, and the new window becomes the master window. When a head becomes to full (ie more windows cannot be placed upon the stack) a overflow policy is used to determine which window to move to a separate head or group.

`gnew-dynamic` *name* [Command]

Create a new dynamic group named NAME.

`gnewbg-dynamic` *name* [Command]

Create a new dynamic group named NAME in the background.

`rotate-windows` *direction* [Command]

Rotate all windows in the current group and head forward (clockwise) or backward (counterclockwise)

`change-layout` *layout* [Command]

Change the layout of the current head and group.

`change-default-layout` *layout* **&optional** (*update-heads* **unset**) [Command]

Change the default layout for dynamic groups.

`change-split-ratio` *ratio* [Command]

Change the size of the master window of the current head and group.

`change-default-split-ratio` *ratio* **&optional** (*update-heads* **unset**) [Command]

Change the default size of the master window for dynamic groups.

`retile` **&optional** (*retile-floats* **t**) [Command]

Force a retile of all windows.

`select-floating-window` **&optional** (*fmt* ***window-format***) *window-list* [Command]

Select a floating window from a menu.

`*rotation-focus-policy*` [Variable]

A keyword determining what frame to focus after rotating the windows in a dynamic group. Valid values are: :PRESERVE, meaning to stay on the same frame :FOLLOW, meaning to follow the current window as it rotates :MASTER, meaning to always stay to the master :MASTER-OR-FOLLOW, meaning to stay on the master, or if initiating the rotation while focused on a stack window to follow that window.

When defining commands, anything restricted to tiling groups will also be active in dynamic groups. To fully restrict it to tiling groups, call the function dyn-blacklist-command on the command in question.

1.3.5 Frame Basics

Frames are the boxes within which windows are displayed. StumpWM starts with a single frame per head, meaning that each monitor shows a single window, full screen. If you want to see windows side-by-side, you can “split” this frame in two, either vertically or horizontally. These frames can be further split, creating nested boxes.

Technically speaking, frames live within a “frame tree”. When you split a frame, the command actually creates *two* new frames side-by-side within the original parent frame. This has implications for the behaviour of many commands that interact with frames.

Within this frame tree model, all frames either contain other frames, or contain windows, or are empty. The command ‘fclear’ will hide all of a frame’s windows and show the background.

1.3.6 Window Basics

Windows are created by programs to display their output. They take the shape of the frame in which they are created. The windows within a frame are ordered by how recently that window was focused. Only the top window in the stack is visible.

1.3.7 System Trays and the Mode Line

Many users choose to sacrifice a little screen real-estate to display some generally useful information: the current time and date, wireless network connections, the names of open windows, etc. StumpWM allows you to display this information in a bar across either the top or the bottom of the screen. There are two ways to do this: using external programs called system trays, or using StumpWM’s own mode line.

System trays are a special kind of X window. They advertise to running programs that they are available for embedding icons or notifications from those programs. They often also display clickable icons for each open window. Common tray programs include the GNOME panel or KDE’s kicker, or simpler programs such as stalonetray. Simply starting one of these programs is usually enough for StumpWM to detect it, place it correctly, and allow it to function normally.

The mode line, a concept borrowed from Emacs, is a built-in part of StumpWM. It is essentially a string of text that can include a variety of information about your current session, including the names of current groups and windows. Several modules provide for different types of information. See Chapter 7 [Mode-line], page 51, (and the modules directory) for more.

1.4 Manipulating Frames and Windows

Frames and windows are concepts borrowed from Emacs and the GNU Screen program, and should be familiar to users of those programs. Others may find the terms a little confusing. In other window managers, a "window" usually refers to a bounded box on the screen, showing output from a single program. StumpWM splits this into two concepts: the "frame" is the bounded box, the "window" is the visible output of a program.

One frame can contain many windows. As new windows are created, they appear at the top of the window-stack of the current frame. This is also a little different from other tiling window managers, many of which automatically create new frames for new windows.

Both frames and windows are ordered by when they were last focused. In the following commands and documentation, the terms "next" and "previous" refer to this order. "Other" refers to the most-recently focused object. Calling "other" commands multiple times will bounce back and forth between the two most recent objects.

By default, StumpWM starts with a single group, called "Default", which contains one full-screen frame per head. You can split individual frames horizontally or vertically using the 'hsplit' and 'vsplit' commands, bound to "C-t S" and "C-t s" by default. When a frame is split, the next-most-recently-focused window is pulled into the new frame. See Chapter 6 [Frames], page 45, and Chapter 5 [Windows], page 35, for a complete listing of commands.

1.4.1 Moving Between Frames

Once you have multiple frames, you can move between them in various ways:

- `fnext` (`C-t o` or `C-t TAB`) jumps to the next frame in the current group's frame list.
- `fother` (`C-t M-TAB`) jumps to the last frame that had focus.
- `fselect` (`C-t f`) displays numbers on each visible frame: hit a number key or click it to move to that frame.
- `move-focus` (`C-t <arrow key>`) focus the frame in the direction of the arrow key pressed.
- `sibling` (unbound by default) focus the frame from which the current frame was split.

1.4.2 Manipulating Windows

Some commands change which window is currently focused, some move windows between frames, and some may do both at once.

There are two general ways to move focus between windows: either between windows belonging to the current frame, or between all windows within the current group. Within a single frame:

- `next-in-frame` (`C-t C-M-n`) focus the next window in the current frame's list of windows.
- `prev-in-frame` (`C-t C-M-p`) focus the previous window in the current frame's list of windows.
- `other-in-frame` (`C-t M-t`) focus the most recently focused window in the current frame's list of windows.
- `frame-windowlist` (unbound by default) display a menu of windows in the currently-focused frame, and allow the user to choose one. Alternately, the command `frame-windows` will simply display the list of window names, with no menu choice available.

Within the current group, the following commands will go straight to the specified window. They will never move a window from its original frame, and so may result in focus switching frames.

- `next` (`C-t M-n`) focus the next window in the current group.
- `prev` (`C-t M-p`) focus the previous window in the current group.
- `other` or `other-window` (unbound by default) focus the most recently focused window in the current group.
- `next-urgent` (`C-t C-u`) focus the next window that has marked itself "urgent".
- `select` or `select-window` (`C-t '`) prompt for the title of a window and focus it. Works with partial completion of the title.
- `select-window-by-name` (unbound by default) prompt for the title of a window and focus it. Requires the window title to be entered exactly.
- `select-window-by-number` (`C-t <number>`) choose a window by number.
- `windowlist` (`C-t "`) display a menu of windows in the currently-focused group, and allow the user to choose one.

The following commands always keep the current frame focused. If the selected window is not in the current frame, it will be pulled there from wherever it is (hence the "pull" naming scheme).

- `pull` or `pull-window-by-number` (`C-t C-<number>`) pull the numbered window into the current frame.
- `pull-hidden-next` (`C-t n` or `C-t SPC`) pull the next currently undisplayed window in the window list into the current frame.
- `pull-hidden-previous` (`C-t p`) pull the previous currently undisplayed window in the window list into the current frame.
- `pull-hidden-other` (`C-t C-t`) pull the most recently focused, currently undisplayed window into the current frame.

The following commands move the current window from one frame to another, bringing focus with them.

- `move-window` (`C-t M-<arrow>`) move the currently focused window in the direction indicated by the arrow key.
- `exchange-direction` (unbound by default) prompt for a direction, then swap the currently focused window with the top window of the frame in that direction.

1.5 Interacting with the Lisp process

Since StumpWM is a Lisp program, there is a way for you to evaluate Lisp code directly, on the same Lisp process that StumpWM is running on. Type `C-t :` and an input box will appear. Then type some Lisp expression.

When you call `eval` this way, you will be in the STUMPWM-USER package, which imports all the exported symbols from the main STUMPWM package.

`*mode-line-border-width*`

Reads the value of **mode-line-border-width**.

`(setf *mode-line-border-width* 3)`
: Sets the variable **mode-line-border-width** to 3.

`(set-prefix-key (kbd "C-M-H-s-z"))`
: Calls the `set-prefix-key` function (and sets a new keyboard prefix)

1.6 Init File

Like other window managers, StumpWM's configuration and startup state can be controlled by an initialization file. Unlike other window managers, StumpWM's init is not limited to changing settings and keybindings. The init file is itself a Common Lisp program running in a Common Lisp environment, so you can write your own hacks and make them a part of your StumpWM experience.

On launch, StumpWM searches for an init file of different names and locations on your system, and will use the first one found in this order:

- '~/.stumpwmrc' is the classic UNIX-style configuration name;
- '~/.stumpwm.d/init.lisp' is an Emacs-style location and name;
- '~/.config/stumpwm/config' is the XDG standard;
- '/etc/stumpwmrc' is a system-wide file giving all users a standardized environment.

StumpWM includes a basic 'sample-stumpwmrc.lisp' in its source directory. You can use this as a template when you're starting out: copy it to the above name and location you prefer and edit it to suit your preferences.

It is possible to split your initialization among multiple files, if you call the additional files from within an init file matching the names and locations listed above.

`*processing-existing-windows*` [Variable]
: True when processing pre-existing windows at startup.

1.7 Contact the StumpWM developers

The StumpWM home page is `http://stumpwm.nongnu.org/`.

The StumpWM mailing list is `stumpwm-devel@nongnu.org` which you can subscribe to at `https://lists.nongnu.org/mailman/listinfo/stumpwm-devel`. It is the preferred way of contacting developers for questions. If you have a bug report or patch, please open an issue or pull request at `https://github.com/stumpwm/stumpwm/issues`.

The StumpWM IRC channel can be found on Freenode at `#stumpwm` (`irc://irc.libera.chat/#stumpwm`).

2 Key Bindings

StumpWM is controlled entirely by keystrokes and Lisp commands. It mimics GNU Screen's keyboard handling. StumpWM's default prefix key is `C-t`.

2.1 List of Default Keybindings

The following is a list of keybindings.

`C-t` *`d`*	Select the window with the corresponding digit *d*
`C-t C-`*`d`*	Pull the window with the corresponding digit *d* into the current frame

`C-t n`
`C-t C-n`
`C-t Space`
	Go to the next window in the window list

`C-t p`
`C-t C-p`
	Go to the previous window in the window list

`C-t '`	Go to a window by name
`C-t "`	Select a window from a list and focus the window.
`C-t C-g`	Abort the current command. This is useful if you accidentally hit `C-t`
`C-t i`	Display information about the current window.
`C-t f`	Select a frame by number
`C-t s`	Split current frame vertically
`C-t S`	Split current frame horizontally

`C-t k`
`C-t C-k`
	Sends a kill message to the current frame and the running program.

`C-t K`	Kills the current frame and running program; like a `kill -9`.

`C-t c`
`C-t C-c`
	Run an X terminal; by default `xterm`

`C-t e`
`C-t C-e`
	Run Emacs or raise it if it is already running.

`C-t t`	Sends a `C-t` to the frame; this is useful for applications like Firefox which make heavy use of `C-t` (in Firefox's case, for opening a new tab). This is similar to how GNU screen uses `C-a a`.

`C-t w`
`C-t C-w`
	Prints out a list of all the windows, their number, and their name.

`C-t RET`
`C-t C-RET`
	Show all windows and let the user select one, make that window the focus.

`C-t b`
`C-t C-b`
	Banish the mouse point to the lower right corner of the screen.

`C-t a`	
`C-t C-a`	Display the current time and date, much like the Unix command `date`.
`C-t C-t`	Switch to the last window to have focus in the current frame.
`C-t !`	Prompt for a shell command to run via `/bin/sh`. All output is discarded.
`C-t R`	If the screen is split into multiple frames, one split will be undone. If there is only one split, the effect will be the same as `C-t Q`.
`C-t o`	
`C-t TAB`	If the screen is split into multiple frames, focus shifts to the `next` frame, where it cycles to the right and then down; analogous to `C-x o` in Emacs.
`C-t F`	Display "Current Frame" in the frame which has focus.
`C-t ;`	Opens the input box. StumpWM commands can be run from here, and the input history moved through.
`C-t :`	Opens the input box, but all things typed in here will be sent to the Common Lisp interpreter where they will be run as Lisp programs; thus, input should be valid Common Lisp.
`C-t C-h`	
`C-t ?`	The help. These can be customized using the **help-keys** variable.
`C-t -`	Hide all frames and show the root window.
`C-t Q`	Removes all splits and maximizes the frame with focus.
`C-t Up`	
`C-t Down`	
`C-t Left`	
`C-t Right`	Shift focus to an adjacent frame in the specified direction. `C-t Up` will shift focus up, if possible, `C-t Down` will shift downwards, etc.
`C-t v`	Prints out the version of the running StumpWM.
`C-t #`	Toggle the mark on the current window
`C-t m`	
`C-t C-m`	Display the last message. Hitting this keybinding again displays the message before that, and so on.
`C-t l`	
`C-t C-l`	redisplay the current window and force it to take up the entire frame.
`C-t G`	Display all groups and windows in each group. For more information see Chapter 8 [Groups], page 57.
`C-t F`*n*	Jump to the corresponding group *n*. `C-t F1` jumps to group 1 and so on.
`C-t g g`	Show the list of groups.
`C-t g c`	Create a new group.

`C-t g n`
`C-t g C-n`
`C-t g SPC`
`C-t g C-SPC`
Go to the next group in the list.

`C-t g N` Go to the next group in the list and bring the current window along.

`C-t g p`
`C-t g C-p` Go to the previous group in the list.

`C-t g P` Go to the previous group in the list and bring the current window along.

`C-t g '` Select a group by name or by number.

`C-t g "` Select a group from a list and switch to it.

`C-t g m` Move the current window to the specified group.

`C-t g k` Kill the current group. All windows are merged into the next group.

`C-t g A`
`C-t g r` Change the current group's name.

`C-t g` *d* Go to the group with digit *d*. `C-t g 1` jumps to group 1 and so on.

`C-t +` Make frames the same height or width in the current frame's subtree.

`C-t h k` Describe the specified key binding.

`C-t h f` Describe the specified function.

`C-t h v` Describe the specified variable.

`C-t h w` List all key sequences that are bound to the specified command

`C-t h c` Describe the specified command.

2.2 Binding Keys

`define-key` *map key command* [Function]

Add a keybinding mapping for the key, *key*, to the command, *command*, in the specified keymap. If *command* is nil, remove an existing binding. For example,

```
(stumpwm:define-key stumpwm:*root-map* (stumpwm:kbd "C-z") "echo Zzzzz...")
```

Now when you type C-t C-z, you'll see the text "Zzzzz..." pop up.

`undefine-key` *map key* [Function]

Clear the key binding in the specified keybinding.

`kbd` *keys* [Function]

This compiles a key string into a key structure used by 'define-key', 'undefine-key', 'set-prefix-key' and others.

`set-prefix-key` *key* [Command]

Change the stumpwm prefix key to KEY.

```
(stumpwm:set-prefix-key (stumpwm:kbd "C-M-H-s-z"))
```

This will change the prefix key to `Control` + `Meta` + `Hyper` + `Super` + the `z` key. By most standards, a terrible prefix key but it makes a great example.

`make-sparse-keymap` [Function]

Create an empty keymap. If you want to create a new list of bindings in the key binding tree, this is where you start. To hang frame related bindings off `C-t C-f` one might use the following code:

```
(defvar *my-frame-bindings*
  (let ((m (stumpwm:make-sparse-keymap)))
    (stumpwm:define-key m (stumpwm:kbd "f") "curframe")
    (stumpwm:define-key m (stumpwm:kbd "M-b") "move-focus left")
    m ; NOTE: this is important
  ))

(stumpwm:define-key stumpwm:*root-map* (stumpwm:kbd "C-f") '*my-frame-bindings*)
```

`*root-map*` [Variable]

This is the keymap by default bound to `C-t` (along with *group-root-map* and either *tile-group-root-map*, *float-group-root-map*, or *dynamic-group-map*). It is known as the *prefix map*.

`*top-map*` [Variable]

The top level key map. This is where you'll find the binding for the *prefix map*.

`*groups-map*` [Variable]

The keymap that group related key bindings sit on. It is bound to `C-t g` by default.

`*group-top-maps*` [Variable]

An alist of the top level maps for each group type. For a given group, all maps whose type matches the given group are active. So for a tile-group, both the group map and tile-group map are active.

Order is important. Each map is seached in the order they appear in the list (inactive maps being skipped). In general the order should go from most specific groups to most general groups.

`*exchange-window-map*` [Variable]

The keymap that exchange-window key bindings sit on. It is bound to `C-t x` by default.

`*key-seq-color*` [Variable]

Color of a keybinding when displayed in windows such as the prefix keybinding in the which-key window.

`bind` *key command* [Command]

Hang a key binding off the escape key.

`unbind` *key* [Command]

Remove a key binding from the escape key.

`send-escape` [Command]

Send the escape key to the current window.

`grab-pointer` *screen* [Function]
: Grab the pointer and set the pointer shape.

`ungrab-pointer` [Function]
: Remove the grab on the cursor and restore the cursor shape.

`*banish-pointer-to*` [Variable]
: Where to put the pointer when no argument is given to (banish-pointer) or the banish command. May be one of :screen :head :frame or :window

`final-key-p` *keys class* [Function]
: Determine if the key is a memeber of a class

`help-key-p` *keys* [Function]
: If the key is for the help command.

`cancel-key-p` *keys* [Function]
: If a key is the cancelling key binding.

`*editor-bindings*` [Variable]
: A list of key-bindings for line editing.

`*numpad-map*` [Variable]
: A keycode to keycode map to re-wire numpads when the numlock key is active

2.3 Modifiers

Many users have had some difficulty with setting up modifiers for StumpWM keybindings. This is caused by a combination of how StumpWM handles modifiers and the default modifiers list for many users' X servers.

- My "Super" key doesn't work!

  This is most likely caused by having the Hyper and Super keys listed as the same modifier in the modifier list.

  ```
  $ xmodmap
  xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

  shift       Shift_L (0x32),  Shift_R (0x3e)
  lock        Caps_Lock (0x42)
  control     Control_L (0x25),  Control_R (0x6d)
  mod1        Alt_L (0x40),  Alt_R (0x71),  Meta_L (0x9c)
  mod2        Num_Lock (0x4d)
  mod3
  mod4        Super_L (0x7f),  Hyper_L (0x80)
  mod5        Mode_switch (0x5d),  ISO_Level3_Shift (0x7c)
  ```

  The problem is in the line beginning with "mod4". The way to set up the modifier list correctly is to have just the Super key as the mod4 modifier. The following `xmodmap` commands will do just that.

  ```
  # clear out the mod4 modifier
  ```

```
$ xmodmap -e 'clear mod4'
$ xmodmap
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x32),  Shift_R (0x3e)
lock        Caps_Lock (0x42)
control     Control_L (0x25),  Control_R (0x6d)
mod1        Alt_L (0x40),  Alt_R (0x71),  Meta_L (0x9c)
mod2        Num_Lock (0x4d)
mod3
mod4
mod5        Mode_switch (0x5d),  ISO_Level3_Shift (0x7c)

# add Super as a mod4 modifier
$ xmodmap -e 'add mod4 = Super_L'
$ xmodmap
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x32),  Shift_R (0x3e)
lock        Caps_Lock (0x42)
control     Control_L (0x25),  Control_R (0x6d)
mod1        Alt_L (0x40),  Alt_R (0x71),  Meta_L (0x9c)
mod2        Num_Lock (0x4d)
mod3
mod4        Super_L (0x73),  Super_L (0x7f)
mod5        Mode_switch (0x5d),  ISO_Level3_Shift (0x7c)
```

You can automate this by storing the commands in a file and calling xmodmap when you start your X session.

```
$ cat ~/.Xmodmap
clear mod4
add mod4 = Super_L
```

If you use `startx`, modify your `~/.xsession` or `~/.xinitrc` file.

```
$ cat ~/.xsession
#!/bin/sh

xmodmap ~/.Xmodmap
exec /usr/bin/stumpwm
```

If you use a settings daemon from one of the major desktop environments (Gnome,KDE, or Unity) you may be able to set keyboard modifiers from their respective configuration GUIs. If not, `xmodmap` should always work if invoked at the right place.

- Handling Meta and Alt: when do I use `M-` and `A-`?

  If you have no Meta keys defined (see the output of the `xmodmap` command), then StumpWM will treat the `M-` prefix in keybindings to mean Alt. However, if there are Meta keys defined, then the `M-` prefix refers to them, and the `A-` prefix refers to Alt.

Most users will simply use `M-` to refer to their Alt keys. However, users that define separate Meta and Alt keys will use `M-` to refer to the former, and `A-` to refer to the latter.

- How can I set up a Hyper key and use it with StumpWM?

  To set up a Hyper key, you need to do two things: bind a physical key to be a Hyper key, and add that key to the modifiers list.

  The following example shows how to bind the control key at the bottom-left of most keyboards to be Hyper. This is useful if you've made Caps Lock into a control key, and have no use for the bottom-left key.

  ```
  $ xmodmap -e 'keycode 37 = Hyper_L'
  $ xmodmap -e 'clear mod5'
  $ xmodmap -e 'add mod5 = Hyper_L'
  ```

  To use a different key for Hyper, replace the keycode "37" above. Use the `xev` program to see the keycode that any physical key has. Refer to the section above on setting up the Super key to see how to automate setting the Hyper key when you start X.

  Now you can use `H-` as a prefix in StumpWM bindings.

  ```
  (define-key *top-map* (kbd "H-RET") "fullscreen")
  (define-key *top-map* (kbd "H-Left") "gprev")
  (define-key *top-map* (kbd "H-Right") "gnext")
  (define-key *top-map* (kbd "H-TAB") "other")
  ```

  Since essentially no programs have Hyper bindings, you can safely bind commands to the **top-map**.

`*all-modifiers*` [Variable]

A list of all keycodes that are considered modifiers

`*modifiers*` [Variable]

A mapping from modifier type to x11 modifier.

2.4 AltGr Keys

StumpWM uses the CLX client library for X11, which doesnt support the XKB extension. As such support for Alt Graph must be handled specially.

Preliminary support for Alt Graph in StumpWM is done by registering AltGr as a modifier key and specially handling it when it is a member of a key press' state. By default StumpWM treats AltGr key presses as a normal key and not a modifier, and this behavior should not be changed unless necessary. To enable treating AltGr as a modifier, the user must place the following in their stumpwmrc:

```
(register-altgr-as-modifier)
```

Additionally, the variable **altgr-offset** defaults to 2, which may be appropriate for the users keyboard layout, or may not. Some users may have an AltGr offset of 4, or potentially 6. If, after calling `register-altgr-as-modifier`, keys on AltGr aren't being registered properly, then the offset may need to be changed. The user can check this explicitly by finding a keysym that requires AltGr and passing it to `keysym-requires-altgr`. As an example, some layouts have ł on `AltGr+l`. In such a case, the following example would return true when **altgr-offset** is correct:

```
(stumpwm::keysym-requires-altgr
  (stumpwm::stumpwm-name->keysym "lstroke"))
```

`register-altgr-as-modifier` [Function]

Register the keysym(s) for ISO_Level3_Shift as modifiers.

`*altgr-offset*` [Variable]

The offset of altgr keysyms. Often 2 or 4, but always an even number.

2.5 Remapped Keys

StumpWM may be configured to translate certain familiar top level keybindings to alternative key sequences that are understood by specific applications. For example, Emacs users are very familiar with `C-n` and `C-p` as keybindings for scrolling down and up one line at a time. However, most applications use these specific keybindings for other actions. The `stumpwm:define-remapped-keys` function may be used to define such application specific remapping of keybindings.

`define-remapped-keys` *specs* [Function]

Define the keys to be remapped and their mappings. The SPECS argument needs to be of the following structure:

(regexp-or-function . (("key-to-remap" . <new-keycodes>) ...))

EXAMPLE: (define-remapped-keys '(("Firefox" ("C-n" . "Down") ("C-p" . "Up") ("C-k" . ("C-S-End" "C-x")))))

The above form remaps Ctrl-n to Down arrow, and Ctrl-p to Up arrow keys. The Ctrl-k key is remapped to the sequence of keys Ctrl-Shift-End followed by Ctrl-x.

```
(define-remapped-keys
    '(("(Firefox|Chrome)"
       ("C-n"   . "Down")
       ("C-p"   . "Up")
       ("C-f"   . "Right")
       ("C-b"   . "Left")
       ("C-v"   . "Next")
       ("M-v"   . "Prior")
       ("M-w"   . "C-c")
       ("C-w"   . "C-x")
       ("C-y"   . "C-v")
       ("M-<"   . "Home")
       ("M->"   . "End")
       ("C-M-b" . "M-Left")
       ("C-M-f" . "M-Right")
       ("M-f"   . "C-Right")
       ("M-b"   . "C-Left")
       ("C-k"   . ("C-S-End" "C-x")))))
```

The above form adds Emacs like keybindings to windows whose *window-class* matches "Firefox" or "Chrome". Additional application specific bindings may be included by using the specific X *window-class* values.

The window matching pattern can also be specified as a function which returns `T` if the focused window matches.

```
;; Match any window with a window-class matching "Firefox"
(define-remapped-keys
    `((,(lambda (win)
          (string-equal "Firefox" (window-class win)))
       ("C-n"   . "Down")
       ("C-p"   . "Up")
       ("C-f"   . "Right")
       ("C-b"   . "Left")
       ("C-v"   . "Next")
       ("M-v"   . "Prior")
       ("M-w"   . "C-c")
       ("C-w"   . "C-x")
       ("C-y"   . "C-v")
       ("M-<"   . "Home")
       ("M->"   . "End")
       ("C-M-b" . "M-Left")
       ("C-M-f" . "M-Right")
       ("M-f"   . "C-Right")
       ("M-b"   . "C-Left")
       ("C-k"   . ("C-S-End" "C-x")))))
```

2.5.1 Circumventing Remapped Keys

However, if the original key binding needs to be explictly applied the `send-raw-key` command may be used. It will prompt for a key which will be passed to the application as-is. For example, if the `send-raw-key` command were bound to `C-t C-q` as follows:

```
(define-key *root-map* (kbd "C-q") "send-raw-key")
```

Then, pressing `C-t C-q`, while the Firefox window has focus, would prompt asking for "Press a key to send". Pressing `C-n` at the prompt will send the keystroke as-is to Firefox, causing it to open a new window.

If more than a single key needs to be passed to the application as-is, the variable **remapped-keys-enabled-p** may be used. Set to nil it will disable all remapped keys.

`send-raw-key` [Command]

Prompts for a key and forwards it to the CURRENT-WINDOW.

`*remapped-keys-enabled-p*` [Variable]

Bool to toggle remapped-keys on/off. Defaults to t

3 Commands

If you've used Emacs before you'll find the distinction between commands and functions familiar. Commands are simply functions that can be bound to keys and executed interactively from StumpWM's input bar. Whereas, in Emacs, the special `"(interactive)"` declaration is used to turn a function into a command, in StumpWM commands are made with a separate `defcommand` or `define-interactive-keymap` macro.

Once a command is defined, you can call it by invoking the `colon` command (`C-t ;`), and typing the name of the command. This may be sufficient for commands that aren't used very often. To see all the currently-defined commands, invoke the command called `commands`: ie press `C-t ;`, type "commands", and hit return.

Commonly-used commands can also be bound to a keystroke, which is much more convenient. To do this, use the `define-key` function (see Chapter 2 [Key Bindings], page 9), giving the name of the command as a string. For example:

```
(define-key *root-map* (kbd "d") "exchange-direction")
```

You cannot give the command name as a symbol, nor can you bind a key to a regular function defined with `defun`.

If the command takes arguments (see Section 3.1 [Writing Commands], page 19), you can fix those arguments when defining the key-binding, by including the arguments in the same string as the command name, separated by a space. For instance, the `exchange-direction` command, which is unbound by default, requires a direction in which to exchange windows. If you call `exchange-direction` directly, it will prompt you for the direction. If you know that you often exchange in left/right directions, and want those actions bound to keys, you can use the following in your customization file:

```
(define-key *root-map* (kbd "[") "exchange-direction left")
(define-key *root-map* (kbd "]") "exchange-direction right")
```

Multiple arguments can be included by adding them to the command string, separated by spaces. Not all argument types can be represented as strings, but StumpWM will do its best to convert types.

StumpWM does not implement the Emacs concept of prefix arguments.

3.1 Writing Commands

StumpWM commands are written much like any Lisp function. The main difference is in the way command arguments are specified. The `defcommand` macro takes a list of arguments as its first form (similar to the `defun` macro), and a corresponding list of types as its second form. All arguments must belong to a "type". Each type specification has two parts: a keyword specifying the argument type, and a string prompt that will be displayed when asking the user to enter the argument value. A typical `defcommand` might look like this:

```
(defcommand now-we-are-six (name age)
    ((:string "Enter your name: ")
     (:number "Enter your age: "))
  (message "~a, in six years you will be ~a" name (+ 6 age)))
```

If `now-we-are-six` is called interactively via the `colon` command, the user will be prompted for a string and a number, which will then be bound to "name" and "age", respectively, in the body of the command.

When invoking the command via a key-binding, it is possible to provide some or all of the arguments directly:

```
(define-key *root-map* (kbd "L") "now-we-are-six John")
```

In this case, hitting `C-t L` will only prompt for an age (the first string argument is already bound to "John"). Argument values provided this way always bind to the earliest arguments defined: ie, it is not possible to specify an age, but prompt the user for a name.

If the type declaration does not include a prompt (ie, it looks like "(:type nil)", or "(:type)" or just ":type"), the argument is considered optional. It can be provided via a key-binding invocation, as above, but if it isn't, the user will not be prompted, and the argument will be bound to nil.

It is possible to limit the scope under which the command will be usable: a command can be defined to work only a specific group type; the three currently implemented are tile groups,f loating groups, and dynamic groups. This is done by replacing the name of the command with a two-element list: the name of the command as a symbol, and either the symbol tile-group or floating-group. For instance, the `next` command, which only functions in tile groups, is defined this way:

```
(defcommand (next tile-group) ...)
```

`defcommand` *name* (**&rest** *args*) (**&rest interactive-args**) **&body** *body* [Macro]

Create a command function and store its interactive hints in *command-hash*. The local variable %interactivep% can be used to check if the command was called interactively. If it is non-NIL then it was called from a keybinding or from the colon command.

The NAME argument can be a string, or a list of two symbols. If the latter, the first symbol names the command, and the second indicates the type of group under which this command will be usable. Currently, tile-group, floating-group and dynamic-group are the possible values.

INTERACTIVE-ARGS is a list of the following form: ((TYPE PROMPT) (TYPE PROMPT) ...)

each element in INTERACTIVE-ARGS declares the type and prompt for the command's arguments.

TYPE can be one of the following:

:y-or-n	A yes or no question returning T or NIL.
:variable	A lisp variable
:function	A lisp function
:command	A stumpwm command as a string.
:key-seq	A key sequence starting from *TOP-MAP*
:window-number	An existing window number
:number	An integer number
:string	A string

:key	A single key chord
:window-name	An existing window's name
:direction	A direction symbol. One of :UP :DOWN :LEFT :RIGHT
:gravity	A gravity symbol. One of :center :top :right :bottom :left :top-right :top-left :bottom-right :bottom-left
:group	An existing group
:frame	A frame
:shell	A shell command
:rest	The rest of the input yet to be parsed.
:module	An existing stumpwm module
:rotation	A rotation symbol. One of :CL, :CLOCKWISE, :CCL, OR :COUNTER-CLOCKWISE

Note that new argument types can be created with DEFINE-STUMPWM-TYPE.

PROMPT can be string. In this case, if the corresponding argument is missing from an interactive call, stumpwm will use prompt for its value using PROMPT. If PROMPT is missing or nil, then the argument is considered an optional interactive argument and is not prompted for when missing.

Alternatively, instead of specifying nil for PROMPT or leaving it out, an element can just be the argument type.

`defcommand-alias` *alias original* [Macro]

Since interactive commands are functions and can conflict with package symbols. But for backwards compatibility this macro creates an alias name for the command that is only accessible interactively.

`*last-command*` [Variable]

Set to the last interactive command run.

3.2 Interactive Keymaps

Interactive keymaps are a special type of command that basically pushes another keymap on top of the current one. The new keymap will only be removed after an exit command is run. An example is `iresize`.

The macro `define-interactive-keymap` is used to define an interactive keymap. The first argument is the same as `defcommand`. The second argument is a list of extra configurations that can be used for controlling the command and the rest are the key bindings for the new command, optionally with a `t` appended; this tells `define-interactive-keymap` to exit the keymap upon use of that keybinding.

For instance, a simple interactive keymap:

```
(define-interactive-keymap my-new-command nil
  ((kbd "a") "execute-a-command")
  ((kbd "b") "execute-b-command" t))
```

This creates a command called `my-new-command` that, when called, will activate the interactive keymap mode. In this mode, the user can press “a” or “b” repeatedly, omitting any prefix. The default exit commands are `RET`, `C-g` and `ESC`.

This creates a command called `my-new-command` that, when called, will activate the interactive keymap mode. In this mode, the user can press “a” or “b”, omitting any prefix. The user can press “a” repeatedly, however pressing “b” exits the keymap. The default exit commands are `RET`, `C-g` and `ESC`.

The available configuration is `on-enter`, `on-exit` and `abort-if`:

```
(defun before-foo () (message "start foo"))
(defun after-foo () (message "end foo"))
(defun foo-p () (and *bar* *baz*))
(defparameter *custom-exit-keys* '((kbd "RET") (kbd "SPC")
                                   (kbd "C-g") (kbd "ESC")))

(define-interactive-keymap foo (:on-enter #'before-foo
                                :on-exit #'after-foo
                                :abort-if #'foo-p
                                :exit-on *custom-exit-keys*))
```

In the above example, the message “start foo” will appear before starting the interactive keymap, “end foo” will appear right after the command exits; We’ve added SPC as an exit key with custom exit keys. Also, the command executes only if the variables `*bar*` and `*baz*` are true.

`define-interactive-keymap` *name* (**&key** *on-enter on-exit abort-if (exit-on (quote ((kbd RET) (kbd ESC) (kbd C-g)))))* **&body** *key-bindings* [Macro]

Declare an interactive keymap mode. This can be used for developing interactive modes or command trees, such as `iresize`.

The NAME argument follows the same convention as in `defcommand`.

ON-ENTER and ON-EXIT are optional functions to run before and after the interactive keymap mode, respectively. If ABORT-IF is defined, the interactive keymap will only be activated if calling ABORT-IF returns true.

KEY-BINDINGS is a list of the following form: ((KEY COMMAND) (KEY COMMAND) ...) If one appends t to the end of a binding like so: ((kbd "n") "cmd" t) then the keymap is immediately exited after running the command.

Each element in KEY-BINDINGS declares a command inside the interactive keymap. Be aware that these commands won’t require a prefix to run.

`call-and-exit-kmap` *command exit-command* [Command]

This command effectively calls two other commands in succession, via run-commands. it is designed for use in the define-interactive-keymap macro, to implement exiting the keymap on keypress.

3.3 StumpWM Types

All command arguments must be of a defined “StumpWM type”. The following types are pre-defined:

:y-or-n	A yes or no question returning T or NIL.
:variable	A lisp variable
:function	A lisp function
:command	A StumpWM command as a string.
:key-seq	A key sequence starting from *TOP-MAP*
:window-number	An existing window number
:number	An integer number
:string	A string
:key	A single key chord
:window-name	An existing window’s name
:direction	A direction symbol. One of :UP :DOWN :LEFT :RIGHT
:gravity	A gravity symbol. One of :center :top :right :bottom :left :top-right :top-left :bottom-right :bottom-left
:group	An existing group
:frame	A frame
:shell	A shell command
:rest	The rest of the input yet to be parsed.
:module	An existing StumpWM module

Additional types can be defined using the macro `define-stumpwm-type`. Emacs users who are accustomed to writing more complicated interactive declarations using "(interactive (list . . .))" forms will find that similar logic can be put into StumpWM type definitions. The macro is called like this:

```
(define-stumpwm-type :type-name (input prompt) body)
```

The keyword :type-name will then be available for use in `defcommand` macros. When commands are called, the bodies of these type definitions are called in turn to produce actual argument values.

Type definitions produce their value in one of several ways: by reading it from the argument line bound to a keystroke, by prompting the user to enter a value, or by generating it programmatically.

Within the body of the type definition, the argument “input” is bound to the argument line provided in the command string, and “prompt” to the string prompt provided in the `defcommand` form. The usual convention is to first check if an argument has been provided in “input” and, if it hasn’t, to prompt for it using “prompt”.

StumpWM provides several convenience functions for handling the value of "input":

- `argument-pop` (input) pop the next space-delimited word or a double quote delimited string argument from the argument line. Backslashes may be used to escape double quotes or backslashes inside double quoted strings.
- `argument-pop-rest` (input) return the remainder of the argument line as a single string, leaving input empty
- `argument-pop-or-read` (input prompt &optional completions) either pop an argument from the argument line, or if it is empty use "prompt" to prompt the user for a value
- `argument-pop-rest-or-read` (input prompt &optional completions) either return the remainder of the argument line as a string, leaving input empty, or use "prompt" to prompt the user for a value

As an example, here's a new type called :smart-direction. The existing :direction type simply asks for one of the four directions "left", "right", "up" or "down", without checking to see if there's a frame in that direction. Our new type, :smart-direction, will look around the current frame, and only allow the user to choose a direction in which another frame lies. If only one direction is possible it will return that automatically without troubling the user. It signals an error for invalid directions; it could alternately return a "nil" value in those cases, and let the command handle that.

```
(define-stumpwm-type :smart-direction (input prompt)
  (let ((valid-dirs
         (loop  ; gather all the directions in which there's a neighbouring frame
            with values = '(("up" :up)
                            ("down" :down)
                            ("left" :left)
                            ("right" :right))
            with frame-set =
              (group-frames (window-group (current-window)))
            for dir in values
            for neighbour = (neighbour
                             (second dir)
                             (window-frame (current-window)) frame-set)
            if (and neighbour (frame-window neighbour))
            collect dir))
        (arg (argument-pop input)))  ; store a possible argument
    (cond ((null valid-dirs)  ; no directions, bail out
           (throw 'error "No valid directions"))
          (arg  ; an arg was bound, but is it valid?
           (or (second (assoc arg valid-dirs :test #'string=))
               (throw 'error "Not a valid direction")))
          ((= 1 (length valid-dirs))  ; only one valid direction
           (second (car valid-dirs)))
          (t  ; multiple possibilities, prompt for direction
           (second (assoc (completing-read input prompt valid-dirs
                                           :require-match t)
                          valid-dirs :test #'string=))))))
```

```
(defcommand smarty (dir) ((:smart-direction "Pick a direction: "))
  ;; 'dir' is a keyword here
  (message "You're going ~a" (string-downcase dir)))

(define-key *root-map* (kbd "R") "smarty right")
```

`define-stumpwm-type` *type* (*input prompt*) **&body** *body* [Macro]

Create a new type that can be used for command arguments. *type* can be any symbol.

When *body* is evaluated *input* is bound to the argument-line. It is passed to `argument-pop`, `argument-pop-rest`, etc. *prompt* is the prompt that should be used when prompting the user for the argument.

```
(define-stumpwm-type :symbol (input prompt)
 (or (find-symbol
       (string-upcase
         (or (argument-pop input)
             ;; Whitespace messes up find-symbol.
             (string-trim " "
                          (completing-read (current-screen)
                                           prompt
                                           ;; find all symbols in the
                                           ;;  stumpwm package.
                                           (let (acc)
                                             (do-symbols (s (find-package "STUMPW
                                               (push (string-downcase (symbol-nam
                                             acc)))
             (throw 'error "Abort.")))
       "STUMPWM")
      (throw 'error "Symbol not in STUMPWM package")))

(defcommand "symbol" (sym) ((:symbol "Pick a symbol: "))
  (message "~a" (with-output-to-string (s)
                  (describe sym s))))
```

This code creates a new type called `:symbol` which finds the symbol in the stumpwm package. The command `symbol` uses it and then describes the symbol.

3.4 Common Built-in Commands

`emacs` [Command]

Start emacs unless it is already running, in which case focus it.

`version` [Command]

Print version information and compilation date.

`banish` **&optional** *where* [Command]

Warp the mouse the lower right corner of the current head.

`ratwarp` *x y* [Command]
: Warp the mouse to the specified location.

`ratrelwarp` *dx dy* [Command]
: Warp the mouse by the specified amount from its current position.

`ratclick` **&optional** (*button* **1**) [Command]
: Simulate a pointer button event at the current pointer location. Note: this function is unlikely to work unless your X server and CLX implementation support XTEST.

`restart-hard` [Command]
: Restart stumpwm. This is handy if a new stumpwm executable has been made and you wish to replace the existing process with it.

  Any run-time customizations will be lost after the restart.

`restart-soft` [Command]
: Soft restart StumpWM. The lisp process isn't restarted. Instead, control jumps to the very beginning of the stumpwm program. This differs from RESTART, which restarts the unix process.

  Since the process isn't restarted, existing customizations remain after the restart.

`lastmsg` [Command]
: Display the last message. If the previous command was lastmsg, then continue cycling back through the message history.

`commands` [Command]
: List all available commands.

`keyboard-quit` [Command]
: This way you can exit from command mode. Also aliased as abort.

`quit` [Command]
: Quit StumpWM.

`reload` [Command]
: Reload StumpWM using `asdf`.

`echo-date` [Command]
: Display the date and time.

`eval-line` *cmd* [Command]
: Evaluate the s-expression and display the result(s).

`command-mode` [Command]
: Command mode allows you to type StumpWM commands without needing the `C-t` prefix. Keys not bound in StumpWM will still get sent to the current window. To exit command mode, type `C-g`.

`list-window-properties` [Command]
: List all the properties of the current window and their values, like xprop.

`show-window-properties` [Command]

Shows the properties of the current window. These properties can be used for matching windows with run-or-raise or window placement rules.

`time` **&rest** *args* [Command]

nil

4 Message and Input Bar

`*suppress-echo-timeout*` [Variable]

Assign this T and messages will not time out. It is recommended to assign this using LET.

`echo` *string* [Command]

Display *string* in the message bar.

`err` *fmt* **&rest** *args* [Function]

run FMT and ARGS through format and echo the result to the current screen along with a backtrace. For careful study, the message does not time out.

`wrap` *words* **&optional** (*max-col* ***message-max-width***) *stream* [Function]

Word wrap at the MAX-COL.

`*message-max-width*` [Variable]

The maximum width of a message before it wraps.

`*help-max-height*` [Variable]

Maximum number of lines for help to display.

`*which-key-format*` [Variable]

The format string that decides how keybindings will show up in the which-key window. Two arguments will be passed to this formatter:

the keybind itself
the associated command

`colon` **&optional** *initial-input* [Command]

Read a command from the user. *initial-text* is optional. When supplied, the text will appear in the prompt.

String arguments with spaces may be passed to the command by delimiting them with double quotes. A backslash can be used to escape double quotes or backslashes inside the string. This does not apply to commands taking :REST or :SHELL type arguments.

`with-restarts-menu` **&body** *body* [Macro]

Execute BODY. If an error occurs allow the user to pick a restart from a menu of possible restarts. If a restart is not chosen, resignal the error.

`restarts-menu` *err* [Function]

Display a menu with the active restarts and let the user pick one. Error is the error being recovered from. If the user aborts the menu, the error is re-signalled.

4.1 Customizing The Bar

The bar's appearance and behavior can be modified with the following functions and variables. See Chapter 15 [Colors], page 89, for an explanation of how to set these color variables.

`set-fg-color` *color* [Function]
: Set the foreground color for the message bar and input bar. *color* can be any color recognized by X.

`set-bg-color` *color* [Function]
: Set the background color for the message bar and input bar. *color* can be any color recognized by X.

`set-border-color` *color* [Function]
: Set the border color for the message bar and input bar. *color* can be any color recognized by X.

`set-msg-border-width` *width* [Function]
: Set the border width for the message bar, input bar and frame indicator.

`set-font` *font* [Function]
: Set the font(s) for the message bar and input bar.

`*message-window-padding*` [Variable]
: The number of pixels that pad the text in the message window.

`*message-window-y-padding*` [Variable]
: The number of pixels that pad the text in the message window vertically.

`*message-window-gravity*` [Variable]
: This variable controls where the message window appears. The following are valid values.

:top-left

:top-right

:bottom-left
:bottom-right
:center

:top

:left

:right

:bottom

`*message-window-input-gravity*` [Variable]
: This variable controls where the message window appears when the input window is being displayed. The following are valid values.

:top-left

:top-right

:bottom-left
:bottom-right
:center

:top

:left

:right

:bottom

`*message-window-timer*` [Variable]

Keep track of the timer that hides the message window.

`*timeout-wait*` [Variable]

Specifies, in seconds, how long a message will appear for. This must be an integer.

`*timeout-wait-multiline*` [Variable]

Specifies, in seconds, how long a message will more than one line will appear for. This must be an integer. If falsy, default to *timeout-wait*.

`*input-window-gravity*` [Variable]

This variable controls where the input window appears. The following are valid values.

:top-left

:top-right

:bottom-left
:bottom-right
:center

:top

:left

:right

:bottom

4.2 Using The Input Bar

The following is a list of keybindings for the Input Bar. Users of Emacs will recognize them.

`DEL` Delete the character before point (`delete-backward-char`).

`M-DEL` Kill back to the beginning of the previous word (`backward-kill-word`).

`C-d`
`Delete` Delete the character after point (`delete-forward-char`).

`M-d` Kill forward to the end of the next word (`forward-kill-word`).

`C-f`
`Right` Move forward one character (`forward-char`).

`M-f` Move forward one word (`forward-word`).

`C-b` `Left`	Move backward one character (`backward-char`).
`M-b`	Move backward one word (`backward-word`).
`C-a` `Home`	Move to the beginning of the current line (`move-beginning-of-line`).
`C-e` `End`	Move to the end of the current line (`move-end-of-line`).
`C-k`	Kill to the end of the line (`kill-line`).
`C-u`	Kill to the beginning of the line (`kill-to-beginning`), the same as `C-a` `C-k`.
`C-p` `Up`	Move to the next earlier entry saved in the command history (`history-back`).
`C-n` `Down`	Move to the next later entry saved in the command history (`history-forward`).
`RET`	Submit the entered command (`submit`).
`C-g`	Abort the current action by closing the Input Bar (`abort`).
`C-y`	Paste text from clipboard into the Input Bar (`yank-selection`).
`TAB`	Clockwise tab complete the current string, if possible. Press `TAB` again to cycle through completions.
`S-TAB`	Counter-clockwise tab complete the current string, if possible. Press `S-TAB` again to cycle through completions.

4.3 Programming The Message Bar

`echo-string` *screen msg* [Function]
Display *string* in the message bar on *screen*. You almost always want to use `message`.

`message` *fmt* **&rest** *args* [Function]
run FMT and ARGS through ‘format’ and echo the result to the current screen.

`with-message-queuing` *new-on-bottom-p* **&body** *body* [Macro]
Queue all messages sent by (MESSAGE ...), (ECHO-STRING ...), (ECHO-STRING-LIST ...) forms within BODY without clobbering earlier messages. When NEW-ON-BOTTOM-P is non-nil, new messages are queued at the bottom.

`*queue-messages-p*` [Variable]
When non-nil, ECHO-STRING-LIST will retain old messages in addition to new ones. When the value is :new-on-bottom, new messages are added to the bottom as in a log file. See also WITH-MESSAGE-QUEUING.

`*input-history-ignore-duplicates*` [Variable]
Do not add a command to the input history if it’s already the first in the list.

`*input-completion-style*` [Variable]

The completion style to use. A completion style has to implement input-completion-reset and input-completion-complete. Available completion styles include

make-input-completion-style-cyclic
make-input-completion-style-unambiguous

`make-input-completion-style-cyclic` [Function]

nil

`make-input-completion-style-unambiguous` **&key** (*display-limit* **64**) [Function]

nil

`copy-last-message` [Command]

Copy the last message displayed into the X selection

4.4 Programming the Input Bar

New input behavior can be added to the input bar by creating editing functions and binding them to keys in the **input-map** using `define-key`, just like other key bindings.

An input function takes 2 arguments: the input structure and the key pressed.

`read-one-line` *screen prompt* **&key** *completions* (*initial-input*) *require-match password* [Function]

Read a line of input through stumpwm and return it. Returns nil if the user aborted.

`read-one-char` *screen* [Function]

Read a single character from the user.

`completing-read` *screen prompt completions* **&key** (*initial-input*) *require-match* [Function]

Read a line of input through stumpwm and return it with TAB completion. Completions can be a list, an fbound symbol, or a function. If its an fbound symbol or a function then that function is passed the substring to complete on and is expected to return a list of matches. If require-match argument is non-nil then the input must match with an element of the completions.

`input-insert-string` *input string* [Function]

Insert *string* into the input at the current position. *input* must be of type *input-line*. Input functions are passed this structure as their first argument.

`input-insert-char` *input char* [Function]

Insert *char* into the input at the current position. *input* must be of type *input-line*. Input functions are passed this structure as their first argument.

`*input-map*` [Variable]

This is the keymap containing all input editing key bindings.

5 Windows

`next` [Command]
: Go to the next window in the window list.

`prev` [Command]
: Go to the previous window in the window list.

`delete-window` **&optional** (*window* **(current-window)**) [Command]
: Delete a window. By default delete the current window. This is a request sent to the window. The window's client may decide not to grant the request or may not be able to if it is unresponsive.

`kill-window` **&optional** (*window* **(current-window)**) [Command]
: Tell X to disconnect the client that owns the specified window. Default to the current window. if `delete-window` didn't work, try this.

`kill-windows-current-group` [Command]
: Kill all windows in the current group.

`kill-windows-other` [Command]
: Kill all windows in current group except the current-window

`echo-windows` **&optional** (*fmt* ***window-format***) (*group* **(current-group)**) (*windows* **(group-windows group)**) [Command]
: Display a list of managed windows. The optional argument *fmt* can be used to override the default window formatting.

`other-window` **&optional** (*group* **(current-group)**) [Command]
: Switch to the window last focused.

`pull-hidden-next` [Command]
: Pull the next hidden window into the current frame.

`pull-hidden-previous` [Command]
: Pull the next hidden window into the current frame.

`pull-hidden-other` [Command]
: Pull the last focused, hidden window into the current frame.

`pull-from-windowlist` **&optional** (*fmt* ***window-format***) [Command]
: Pulls a window selected from the list of windows. This allows a behavior similar to Emacs' switch-to-buffer when selecting another window.

`renumber` *nt* **&optional** (*group* **(current-group)**) [Command]
: Change the current window's number to the specified number. If another window is using the number, then the windows swap numbers. Defaults to current group.

`meta` *key* [Command]
: Send a fake key to the current window. *key* is a typical StumpWM key, like `C-M-o`.

`select-window` *query* [Command]
: Switch to the first window that starts with *query*.

`select-window-by-number` *num* **&optional** (*group* **(current-group)**) [Command]
: Find the window with the given number and focus it in its frame.

`select-window-by-name` *name* [Command]
: Switch to the first window whose name is exactly *name*.

`repack-window-numbers` **&optional** *preserved* [Command]
: Ensure that used window numbers do not have gaps; ignore PRESERVED window numbers.

`title` *title* [Command]
: Override the current window's title.

`windowlist` **&optional** (*fmt* ***window-format***) *window-list* [Command]
: Allow the user to select a window from the list of windows and focus the selected window. For information of menu bindings see Section 14.1 [Menus], page 82. The optional argument *fmt* can be specified to override the default window formatting. The optional argument *window-list* can be provided to show a custom window list (see `windowlist-by-class`). The default window list is the list of all window in the current group. Also note that the default window list is sorted by number and if the *windows-list* is provided, it is shown unsorted (as-is).

`windowlist-by-class` **&optional** (*fmt* ***window-format-by-class***) [Command]
: Allow the user to select a window from the list of windows (sorted by class) and focus the selected window. For information of menu bindings see Section 14.1 [Menus], page 82. The optional argument *fmt* can be specified to override the default window formatting. This is a simple wrapper around the command `windowlist`.

`fullscreen` [Command]
: Toggle the fullscreen mode of the current widnow. Use this for clients with broken (non-NETWM) fullscreen implementations, such as any program using SDL.

`info` **&optional** (*fmt* ***window-info-format***) [Command]
: Display information about the current window.

`refresh` [Command]
: Refresh current window without changing its size.

`redisplay` [Command]
: Refresh current window by a pair of resizes, also make it occupy entire frame.

`float-this` [Command]
: Transforms a tile-window into a float-window

`unfloat-this` [Command]
: Transforms a float-window into a tile-window

`flatten-floats` [Command]

Transform all floating windows in this group to tiled windows. Puts all tiled windows in the first frame of the group.

`unmaximize` **&optional** (*window* **(current-window)**) [Command]

Use the size the program requested for current window (if any) instead of maximizing it.

`toggle-always-on-top` [Command]

Toggle whether the current window always appears over other windows. The order windows are added to this list determines priority.

`toggle-always-show` [Command]

Toggle whether the current window is shown in all groups.

`window-send-string` *string* **&optional** (*window* **(current-window)**) [Command]

Send the string of characters to the current window as if they'd been typed.

`window-head` *window* [Function]

Report what window the head is currently on.

`window-sync` *window what-changed* [Function]

Some window slot has been updated and the window may need to sync itself. WHAT-CHANGED is a hint at what changed.

`window-visible-p` *window* [Function]

Return T if the window is visible

`raise-window` *window* [Function]

Bring the window to the top of the window stack.

`focus-window` *window* **&optional** *raise* [Function]

Give the specified window keyboard focus and (optionally) raise.

`*xwin-to-window*` [Variable]

Hash table for looking up windows quickly.

`*window-format*` [Variable]

This variable decides how the window list is formatted. It is a string with the following formatting options:

%n	Substitutes the window's number translated via *window-number-map*, if there are more windows than *window-number-map* then will use the window-number.
%s	Substitute the window's status. * means current window, `+` means last window, and - means any other window.
%t	Substitute the window's name.
%c	Substitute the window's class.
%i	Substitute the window's resource ID.

%m Draw a # if the window is marked.

Note, a prefix number can be used to crop the argument to a specified size. For instance, '`%20t`' crops the window's title to 20 characters.

`*window-info-format*` [Variable]

The format used in the info command. See **window-format** for formatting details.

`*window-name-source*` [Variable]

This variable controls what is used for the window's name. The default is `:title`.

`:title` Use the window's title given to it by its owner.

`:class` Use the window's resource class.

`:resource-name`
Use the window's resource name.

`*new-window-preferred-frame*` [Variable]

This variable controls what frame a new window appears in. It is a list of preferences. The first preference that is satisfied is used. Valid list elements are as follows:

`:focused` Choose the focused frame.

`:last` Choose the last focused frame.

`:empty` Choose any empty frame.

`:unfocused`
Choose any unfocused frame.

Alternatively, it can be set to a function that takes one argument, the new window, and returns the preferred frame or a list of the above preferences.

`*hidden-window-color*` [Variable]

Color command for hidden windows when using the fmt-head-window-list-hidden-windows formatter. To disable coloring hidden windows, set this to an empty string.

`*honor-window-moves*` [Variable]

Allow windows to move between frames.

`define-fullscreen-in-frame-rule` *name* (*window-argument*) **&body** *body* [Macro]

Define a rule for a window to be fullscreened within the frame. Each rule is a function which will be called when a window is made fullscreen. If the rule returns NIL then the fullscreen window takes up the entire head, otherwise it takes up only its frame. Within the body of the rule *WINDOW-ARGUMENT* is bound to the window being processed.

`add-fullscreen-in-frame-rule` *name function* **&key** *shadow* [Function]

Add a function to the fullscreen-in-frame window rules alist. If *NAME* already exists as a key in the alist and *SHADOW* is nil, then *FUNCTION* replaces the existing value. Otherwise *NAME* and *FUNCTION* are pushed onto the alist.

`remove-fullscreen-in-frame-rule` *name* **&key** *count* [Function]

Remove rules named *NAME* from the fullscreen-in-frame window rules alist. If *COUNT* is NIL then all matching rules are removed, otherwise only the first *COUNT* rules are removed.

`*fullscreen-in-frame-p-window-functions*` [Variable]

A alist of predicate functions for determining if a window should be fullscreen in frame.

5.1 Window Marks

Windows can be marked. A marked window has a # beside it in the window list. Some commands operate only on marked windows.

`mark` [Command]

Toggle the current window's mark.

`clear-window-marks` **&optional** (*group* **(current-group)**) (*windows* **(group-windows group)**) [Command]

Clear all marks in the current group.

`pull-marked` [Command]

Pull all marked windows into the current frame and clear the marks.

5.2 Customizing Window Appearance

`*maxsize-border-width*` [Variable]

The width in pixels given to the borders of windows with maxsize or ratio hints.

`*transient-border-width*` [Variable]

The width in pixels given to the borders of transient or pop-up windows.

`*normal-border-width*` [Variable]

The width in pixels given to the borders of regular windows.

`*window-border-style*` [Variable]

This controls the appearance of the border around windows. valid values are:

:thick	All space within the frame not used by the window is dedicated to the border.
:thin	Only the border width as controlled by *maxsize-border-width* *normal-border-width* and *transient-border-width* is used as the border. The rest is filled with the unfocus color.
:tight	The same as :thin but the border surrounds the window and the wasted space within the frame is not obscured, revealing the background.
:none	Like :tight but no border is ever visible.

After changing this variable you may need to call sync-all-frame-windows to see the change.

See Chapter 15 [Colors], page 89, for an explanation of how to set these color variables.

`set-win-bg-color` *color* [Function]

Set the background color of the window. The background color will only be visible for windows with size increment hints such as '`emacs`' and '`xterm`'.

`set-focus-color` *color* [Function]

Set the border color for focused windows. This is only used when there is more than one frame.

`set-unfocus-color` *color* [Function]

Set the border color for windows without focus. This is only used when there is more than one frame.

`set-float-focus-color` *color* [Function]

Set the border color for focused windows in a float group.

`set-float-unfocus-color` *color* [Function]

Set the border color for windows without focus in a float group.

`set-normal-gravity` *gravity* [Function]

Set the default gravity for normal windows. Possible values are `:center` `:top` `:left` `:right` `:bottom` `:top-left` `:top-right` `:bottom-left` and `:bottom-right`.

`set-maxsize-gravity` *gravity* [Function]

Set the default gravity for maxsize windows.

`set-transient-gravity` *gravity* [Function]

Set the default gravity for transient/pop-up windows.

`gravity` *gravity* [Command]

Set a window's gravity within its frame. Gravity controls where the window will appear in a frame if it is smaller that the frame. Possible values are:

center

top

right

bottom

left

top-right

top-left

bottom-right
bottom-left

`gravity-coords` *gravity width height minx miny maxx maxy* [Function]

Get the X and Y coordinates to place something of width WIDTH and height HEIGHT within an area defined by MINX MINY MAXX and MAXY, guided by GRAVITY.

5.3 Controlling Raise And Map Requests

It is sometimes handy to deny a window's request to be focused. The following variables determine such behavior.

A map request occurs when a new or withdrawn window requests to be mapped for the first time.

A raise request occurs when a client asks the window manager to give an existing window focus.

`*deny-map-request*` [Variable]
: A list of window properties that stumpwm should deny matching windows' requests to become mapped for the first time.

`*deny-raise-request*` [Variable]
: Exactly the same as **deny-map-request** but for raise requests.
: Note that no denial message is displayed if the window is already visible.

`*suppress-deny-messages*` [Variable]
: For complete focus on the task at hand, set this to `T` and no raise/map denial messages will be seen.

Some examples follow.

```
;; Deny the firefox window from taking focus when clicked upon.
(push '(:class "gecko") stumpwm:*deny-raise-request*)

;; Deny all map requests
(setf stumpwm:*deny-map-request* t)

;; Deny transient raise requests
(push '(:transient) stumpwm:*deny-map-request*)

;; Deny the all windows in the xterm class from taking focus.
(push '(:class "Xterm") stumpwm:*deny-raise-request*)
```

5.4 Programming With Windows

`define-window-slot` *attr* [Macro]
: Create a new window attribute and corresponding get/set functions.

`window-send-string` *string* **&optional** (*window* **(current-window)**) [Function]
: Send the string of characters to the current window as if they'd been typed.

`*default-window-name*` [Variable]
: The name given to a window that does not supply its own name.

`*window-events*` [Variable]
: The events to listen for on managed windows.

`*window-parent-events*` [Variable]
: The events to listen for on managed windows' parents.

5.5 Rule Based Window Placement

`define-frame-preference` *target-group* **&body** *frame-rules* [Macro]

Create a rule that matches windows and automatically places them in a specified group and frame. Each frame rule is a lambda list:

```
(frame-number raise lock &key from-group create restore dump-name class class-not
instance instance-not type type-not role role-not title title-not
match-properties-and-function match-properties-or-function)
```

target-group
: When nil, rule applies in the current group. When non nil, *lock* determines applicability of rule

frame-number
: The frame number to send matching windows to

raise
: When non-nil, raise and focus the window in its frame

lock
: When this is nil, this rule will only match when *target-group* matches the group designated by *from-group*. When non-nil, this rule matches regardless of the group and the window is sent to *target-group*. If *lock* and *raise* are both non-nil, then stumpwm will jump to the specified group and focus the matched window.

from-group
: When *lock* is NIL, and this is non-NIL, this rule will only match when *target-group* matches *from-group*. This should be set to either a group name(a string), or an expression that returns a group(e.g (current-group)). When this is NIL, the rule matches if *target-group* matches the group the window is in, or the current group if the window has no group.

create
: When non-NIL the group is created and eventually restored when the value of create is a group dump filename in *DATA-DIR*. Defaults to NIL.

restore
: When non-NIL the group is restored even if it already exists. This arg should be set to the dump filename to use for forced restore. Defaults to NIL

class
: The windows class must match *class*.

class-not
: The windows class must not match *class-not*

instance
: The windows instance/resource name must match *instance*.

instance-not
: The windows instance/resource name must not match *instance-not*.

type
: The windows type must match *type*.

type-not
: The windows type must not match *type-not*.

role
: The windows role must match *role*.

role-not
: The windows role must not match *role-not*.

title The windows title must match *title*.

title-not The windows title must not match *title-not*.

match-properties-and-function
: A function that, if provided, must return true alongside the provided properties in order for the rule to match. This function takes one argument, the window. Must be an unquoted symbol to be looked up at runtime.

match-properties-or-function
: A function that, if provided and returning true, will cause the rule to match regardless of whether the window properties match. Takes one argument, the window. Must be an unquoted symbol to be looked up at runtime.

`clear-window-placement-rules` [Function]
Clear all window placement rules.

`remember` *lock title* [Command]
Make a generic placement rule for the current window. Might be too specific/not specific enough!

`forget` [Command]
Forget the window placement rule that matches the current window.

`dump-window-placement-rules` *file* [Command]
Dump *window-placement-rules* to FILE.

`restore-window-placement-rules` *file* [Command]
Restore *window-placement-rules* from FILE.

`*window-placement-rules*` [Variable]
List of rules governing window placement. Use define-frame-preference to add rules

5.6 Window Selection Expressions

Window Selection Expressions (WSE) were inspired by SQL. The intent is to allow writing consise code to select the windows you need and to act upon them (or just to get the list of selected windows). The implementation includes a set of (hopefully) consistent concisely-named wrappers for the StumpWM functionality useful for window set description and the act-on-matching-windows macro that encapsulates the logic of iterating over a window set.

If we had SQL in StumpWM, we would write `select window_id from windows as w where w.title = 'XTerm'`. WSE chooses to be more Lisp-style and instead uses `(act-on-matching-windows (w) (titled-p w "XTerm") w)`

The `act-on-matching-windows` function also allows performing some actions, for example getting all the windows titled XTerm into the current group: `(act-on-matching-windows (w) (titled-p w "XTerm") (pull-w w))`

`move-windows-to-group` *windows* **&optional** (*arggroup* **nil**) [Function]
Move all windows from the list to the group

`act-on-matching-windows` (*var* **&optional** *(range (quote (current-screen))))* *condition* **&rest** *code* [Macro]

Run code on all windows matching condition; var is the shared lambda variable. Range can be any screen/group/frame or :screen/:group/:frame for the current instance. Condition is just the code to evaluate.

`pull-w` *w* **&optional** *g* [Function]

Pull the window w: to the current group or to the specified group g.

`titled-p` *w title* [Function]

Check whether window title of the window w is equal to the string title.

`title-re-p` *w tre* [Function]

Check whether the window title of the window w matches the regular expression tre.

`classed-p` *w class* [Function]

Check whether the window class of the window w is equal to the string class.

`class-re-p` *w cre* [Function]

Check whether the window class of the window w matches the regular expression cre.

`typed-p` *w type* [Function]

Check whether the window type of the window w is equal to the string type.

`type-re-p` *w tre* [Function]

Check whether the window type of the window w matches the regular expression tre.

`roled-p` *w role* [Function]

Check whether the window role of the window w is equal to the string role.

`role-re-p` *w rre* [Function]

Check whether the window role of the window w matches the regular expression rre.

`resed-p` *w res* [Function]

Check whether the window resource of the window w is equal to the string res.

`res-re-p` *w rre* [Function]

Check whether the window resource of the window w matches the regular expression rre.

`grouped-p` *w* **&optional** *name* [Function]

Check whether the window w belongs to the group name or the current group if name is not specified.

`in-frame-p` *w* **&optional** *f* [Function]

Check whether the window w belongs to the frame f or to the current frame if the frame is not specified.

6 Frames

Frames contain windows. All windows exist within a frame.

Those used to ratpoison will notice that this differs from ratpoison's window pool, where windows and frames are not so tightly connected.

`pull-window-by-number` *n* **&optional** (*group* **(current-group)**) [Command]
: Pull window N from another frame into the current frame and focus it.

`hsplit` **&optional** (*ratio* **1/2**) [Command]
: Split the current frame into 2 side-by-side frames.

`vsplit` **&optional** (*ratio* **1/2**) [Command]
: Split the current frame into 2 frames, one on top of the other.

`hsplit-equally` *amt* [Command]
: Deprecated. Use 'vsplit-uniformly' instead.

`vsplit-uniformly` *amt* [Command]
: Split current frame in n rows of equal size.

`vsplit-equally` *amt* [Command]
: Deprecated. Use 'hsplit-uniformly' instead.

`hsplit-uniformly` *amt* [Command]
: Split current frame in n columns of equal size.

`remove-split` **&optional** (*group* **(current-group)**) (*frame* **(tile-group-current-frame group)**) [Command]
: Remove the specified frame in the specified group (defaults to current group, current frame). Windows in the frame are migrated to the frame taking up its space.

`only` [Command]
: Delete all the frames but the current one and grow it to take up the entire head.

`curframe` [Command]
: Display a window indicating which frame is focused.

`fnext` [Command]
: Cycle through the frame tree to the next frame.

`fprev` [Command]
: Cycle through the frame tree to the previous frame.

`sibling` [Command]
: Jump to the frame's sibling. If a frame is split into two frames, these two frames are siblings.

`fother` [Command]
: Jump to the last frame that had focus.

`fselect` *frame-number* [Command]
: Display a number in the corner of each frame and let the user to select a frame by number or click. If *frame-number* is specified, just jump to that frame.

`resize` *width height* [Command]
: Move the frame split directly to the right of the current frame as much as possible up to *width* pixels, or if impossible try the split directly to the left instead. Similarly, also move the frame split directly below the current frame as much as possible up to *height* pixels, or if impossible try the split directly above instead.

`resize-direction` *d* [Command]
: Resize frame to direction *d*

`balance-frames` **&aux** (*group* **(current-group)**) [Command]
: Make frames the same height or width in the current frame's subtree.

`fclear` [Command]
: Clear the current frame.

`move-focus` *dir* [Command]
: Focus the frame adjacent to the current one in the specified direction. The following are valid directions:

  up

  down

  left

  right

`move-window` *dir* [Command]
: Just like move-focus except that the current is pulled along.

`next-in-frame` [Command]
: Go to the next window in the current frame.

`prev-in-frame` [Command]
: Go to the previous window in the current frame.

`other-in-frame` [Command]
: Go to the last accessed window in the current frame.

`next-urgent` [Command]
: Jump to the next urgent window

`frame-windowlist` **&optional** (*fmt* ***window-format***) [Command]
: Allow the user to select a window from the list of windows in the current frame and focus the selected window. The optional argument *fmt* can be specified to override the default window formatting.

`echo-frame-windows` **&optional** (*fmt* ***window-format***) [Command]
: Display a list of all the windows in the current frame.

`exchange-direction` *dir* **&optional** (*win* **(current-window)**) [Command]

Exchange the current window (by default) with the top window of the frame in specified direction. (bound to `C-t x` by default)

up

down

left

right

`expose` [Command]

Automagically tile all windows and let the user select one, make that window the focus. Set the variable '*expose-auto-tile-fn*' to another tiling function if a different layout is desired. Set '*expose-n-max*' to the maximum number of windows to be displayed for choosing.

`save-frame-excursion` **&body** *body* [Macro]

Execute body and then restore the current frame.

`run-or-pull` *cmd props* **&optional** (*all-groups* ***run-or-raise-all-groups***) (*all-screens* ***run-or-raise-all-screens***) [Function]

Similar to run-or-raise, but move the matching window to the current frame instead of switching to the window.

`only-one-frame-p` [Function]

T if there is only one maximized frame in the current head. This can be used around a the "only" command to avoid the warning message.

`*min-frame-width*` [Variable]

The minimum width a frame can be. A frame will not shrink below this width. Splitting will not affect frames if the new frame widths are less than this value.

`*min-frame-height*` [Variable]

The minimum height a frame can be. A frame will not shrink below this height. Splitting will not affect frames if the new frame heights are less than this value.

`*new-frame-action*` [Variable]

When a new frame is created, this variable controls what is put in the new frame. Valid values are

`:empty` The frame is left empty

`:last-window`
The last focused window that is not currently visible is placed in the frame. This is the default.

`*expose-auto-tile-fn*` [Variable]

Function to call to tile current windows.

`*expose-n-max*` [Variable]

Maximum number of windows to display in the expose `*frame-indicator-text*` [Variable]

What appears in the frame indicator window?

`*frame-indicator-timer*` [Variable]

Keep track of the timer that hides the frame indicator.

`*frame-number-map*` [Variable]

Set this to a string to remap the frame numbers to more convenient keys. For instance,

"hutenosa"

would map frame 0 to 7 to be selectable by hitting the appropriate homerow key on a dvorak keyboard. Currently, only single char keys are supported. By default, the frame labels are the 36 (lower-case) alphanumeric characters, starting with numbers 0-9.

6.1 Interactively Resizing Frames

There is a mode called `iresize` that lets you interactively resize the current frame. To enter the mode use the `iresize` command or type `C-t r`.

The following keybindings apply to the mode:

`C-p`
`Up`
`k` Shrink the frame vertically.

`C-n`
`Down`
`j` Expand the frame vertically.

`C-f`
`Right`
`l` Expand the frame horizontally.

`C-b`
`Left`
`h` Shrink the frame horizontally.

`C-g`
`ESC` Abort the interactive resize.

`RET` Select the highlighted option.

`iresize` [Command]

Starts interactive command "IRESIZE"

`setup-iresize` [Function]

Start the interactive resize mode.

`*resize-map*` [Variable]

The keymap used for resizing a window

`*resize-increment*` [Variable]

Number of pixels to increment by when interactively resizing frames.

6.2 Frame Dumping

The configuration of frames and groups can be saved and restored using the following commands.

`dump-desktop-to-file` *file* [Command]

Dumps the frames of all groups of all screens to the named file. If FILE is an absolute path, then the dump will be read written there. Otherwise, defaults to writing to "FILE.dump" in the XDG_DATA_HOME location.

`dump-group-to-file` *file* [Command]

Dumps the frames of the current group of the current screen to the named file. If FILE is an absolute path, then the dump will be read written there. Otherwise, defaults to writing to "FILE.dump" in the XDG_DATA_HOME location.

`dump-screen-to-file` *file* [Command]

Dumps the frames of all groups of the current screen to the named file. If FILE is an absolute path, then the dump will be read written there. Otherwise, defaults to writing to "FILE.dump" in the XDG_DATA_HOME location.

`restore-from-file` *file* [Command]

Restores screen, groups, or frames from named file, depending on file's contents. If FILE is an absolute path, then the dump will be read from there. Otherwise, defaults to reading from "FILE.dump" in the XDG_DATA_HOME location.

`place-existing-windows` [Command]

Re-arrange existing windows according to placement rules.

`place-current-window` [Command]

Re-arrange current window according to placement rules.

The configuration files are stored in the *$XDG_CONFIG_HOME/stumpwm*. The file name specified is saved as a `.dump` file type. For example, `: dump-desktop-to-file example` may save a file in `~/.local/share/stumpwm/example.dump`. `restore-from-file` also adds the `.dump` extension by default.

7 The Mode Line

The mode line is a bar that runs across either the top or bottom of a head and is used to display information. By default the mode line displays the list of windows, similar to the output `C-t w` produces.

Alternatively, external panel applications such as the GNOME panel and KDE's kicker may be used. Simply starting one of these programs is enough to set it as the mode line of the head it would like to be on (if the panel is XRandR aware) or whichever head is available. In order to avoid problems displaying menus, configure your panel application for positioning at the top or bottom of the head rather than relying on **mode-line-position**

The mode line can be turned on and off with the `mode-line` command or the lisp function `stumpwm:toggle-mode-line`. Each head has its own mode line. For example:

```
;; turn on/off the mode line for the current head only.
(stumpwm:toggle-mode-line (stumpwm:current-screen)
                          (stumpwm:current-head))
```

The mode line is updated after every StumpWM command.

To display the window list and the current date on the modeline, one might do the following:

```
(setf stumpwm:*screen-mode-line-format*
      (list "%w | "
            '(:eval (stumpwm:run-shell-command "date" t))))
```

`(stumpwm:run-shell-command "date" t)` runs the command `date` and returns its output as a string.

`mode-line` [Command]

A command to toggle the mode line visibility.

`toggle-mode-line` *screen head* **&optional** (*format* **(quote *screen-mode-line-format*)**) [Function]

Toggle the state of the mode line for the specified screen

`*screen-mode-line-format*` [Variable]

This variable describes what will be displayed on the modeline for each screen. Turn it on with the function TOGGLE-MODE-LINE or the mode-line command.

It is a list where each element may be a string, a symbol, or a list.

For a symbol its value is used.

For a list of the form (:eval FORM) FORM is evaluated and the result is used as a mode line element.

If it is a string the string is printed with the following formatting options:

%h	List the number of the head the mode-line belongs to
%w	List all windows in the current group windows using **window-format**
%W	List all windows on the current head of the current group using **window-format**
%g	List the groups using **group-format**

%n	The current group's name
%u	Using **window-format**, return a 1 line list of the urgent windows, space seperated.
%v	Using **window-format**, return a 1 line list of the windows, space separated. The currently focused window is highlighted with fmt-highlight. Any non-visible windows are colored the *hidden-window-color*.
%d	Using **time-modeline-string**, print the time.

A number of modules have been written that extends the possible formatting strings. See their documentation for details.

`*time-format-string-default*` [Variable]
The default value for 'echo-date', (e.g, Thu Mar 3 2005 23:05:25).

`*time-modeline-string*` [Variable]
The default time value to pass to the modeline.

`*new-mode-line-hook*` [Hook]
Called whenever the mode-line is created. It is called with argument, the mode-line

`*screen-mode-line-formatters*` [Variable]
An alist containing format character format function pairs for formatting screen mode-lines. functions are passed the mode line.

`*window-formatters*` [Variable]
an alist containing format character format function pairs for formatting window lists.

`bar` *percent width full empty* [Function]
Return a progress bar string of WIDTH characters composed of characters FULL and EMPTY at PERCENT complete.

`bar-zone-color` *amount* **&optional** (*med* **20**) (*hi* **50**) (*crit* **90**) *reverse* [Function]
Return a color command based on the magnitude of the argument. If the limits for the levels aren't specified, they default to sensible values for a percentage. With reverse, lower numbers are more critical.

`add-screen-mode-line-formatter` *character fmt-fun* [Function]
Add a format function to a format character (or overwrite an existing one).

`enable-mode-line` *screen head state* **&optional** *format* [Function]
Set the state of SCREEN's HEAD's mode-line. If STATE is T and FORMAT is specified, then the mode-line's format is updated.

The following variables control the color, position, and size of the mode line. See Chapter 15 [Colors], page 89, for an explanation of how to set these color variables.

`*mode-line-position*` [Variable]
Specifies where the mode line is displayed. Valid values are :top and :bottom.

`*mode-line-border-width*` [Variable]
Specifies how thick the mode line's border will be. Integer value.

`*mode-line-highlight-template*` [Variable]
: The string passed to FORMAT to highlight things in the mode line.

`*mode-line-pad-x*` [Variable]
: Specifies the number of padding pixels between the text and the side of the mode line. Integer value.

`*mode-line-pad-y*` [Variable]
: The number of padding pixels between the modeline text and the top/bottom of the modeline. Integer value.

`*mode-line-background-color*` [Variable]
: The mode line background color.

`*mode-line-foreground-color*` [Variable]
: The mode line foreground color.

`*mode-line-border-color*` [Variable]
: The mode line border color.

`*mode-line-timeout*` [Variable]
: The modeline updates after each command, when a new window appears or an existing one disappears, and on a timer. This variable controls how many seconds elapse between each update. If this variable is changed while the modeline is visible, you must toggle the modeline to update timer.

7.1 Mode-line Interaction

Mode line formatters can register sections of text to be clickable by use of the color formatters `:on-click` and `:on-click-end`. Any text enclosed by these formatters has its bounds saved, and when the mode line recieves a button press event these bounds are checked against to find a clickable area, whose registered function is then called. These formatters can be thought of as similar to XML tags.

To disable the on-click behavior, remove the function `check-for-ml-press` from the hook **mode-line-click-hook**.

To call a function by click the function must first be registered. The function must take at least one argument, the button code. Here is an example of a click-to-focus function and its registration:

```
(defun ml-on-click-focus-window (code id &rest rest)
  (declare (ignore code rest))
  (when-let ((window (window-by-id id)))
    (focus-all window)))

(register-ml-on-click-id :ml-on-click-focus-window #'ml-on-click-focus-window)
```

This defines a function that focuses a window based upon its X11 window ID, and registers it under the ID `:ml-on-click-focus-window`. Here is an example of a mode line formatter that makes use of this function:

```
(add-screen-mode-line-formatter #\i 'fmt-head-window-list-clickable)
```

```
(defun fmt-head-window-list-clickable (ml)
  "Using *window-format*, return a 1 line list of the windows,
space seperated and clickable."
  (flet ((fmt-w (w)
           (let ((str (format-expand *window-formatters*
                                     *window-format*
                                     w)))
             (format-with-on-click-id (if (eq w (current-window))
                                          (fmt-highlight str)
                                          str)
                                      :ml-on-click-focus-window
                                      (window-id w)))))
    (format nil "~{~a~^ ~}"
            (mapcar #'fmt-w
                    (sort1 (head-windows (mode-line-current-group ml)
                                         (mode-line-head ml))
                           #'< :key #'window-number)))))
```

In the above formatter, every windows expansion is wrapped in a :on-click/end pair, which takes the ID we registered as the function to call and the window ID as an argument to be passed to its function. The arguments provided to `:on-click` will be read but not evaluated. The string generated will look like so:

```
"^(:on-click :ml-on-click-focus-window 308242)window text^(:on-click-end)"
```

Clickable text can be nested, in which case the innermost clickable text will take precendent. In the following example `:id2` will be dispatched when clicking `2`, but `:id1` will be dispatched when clicking `1` and `3`:

```
"^(:on-click :id1)1^(:on-click :id2)2^(:on-click-end)3^(:on-click-end)"
```

If one wished for right click to delete windows, then the following example could be placed in the .stumpwmrc:

```
(labels ((ml-on-click-focus-or-delete-window (code id &rest rest)
           (declare (ignore rest))
           (when-let ((window (window-by-id id)))
             (let ((button (decode-button-code code)))
               (case button
                 ((:left-button)
                  (focus-all window))
                 ((:right-button)
                  (delete-window window)))))))
  (register-ml-on-click-id :ml-on-click-focus-window
                           #'ml-on-click-focus-or-delete-window))
```

This will replace the `:ml-on-click-focus-window` function, and all uses of `:on-click` formatters referring to `:ml-on-click-focus-window` will use the new function.

`register-ml-on-click-id` *id fn* [Function]

Register FN with ID, to be used by the :on-click mode line color formatter.

`format-with-on-click-id` *string id* **&rest** *arguments* [Function]

Wrap STRING in :on-click and :on-click-end color formatters, using ID as the id to call when clicked and ARGUMENTS as the arguments to pass to the ID's function. STRING may not contain the :> color formatter, but may contain any other color formatters.

8 Groups

Groups in StumpWM are more commonly known as *virtual desktops* or *workspaces*. Why not create a new term for it?

`gnew` *name* [Command]
: Create a new group with the specified name. The new group becomes the current group. If *name* begins with a dot ("."") the group new group will be created in the hidden state. Hidden groups have group numbers less than one and are invisible to from gprev, gnext, and, optionally, groups and vgroups commands.

`gnew-float` *name* [Command]
: Create a floating window group with the specified name and switch to it.

`gnew-dynamic` *name* [Command]
: Create a new dynamic group named NAME.

`gnewbg` *name* [Command]
: Create a new group but do not switch to it.

`gnewbg-float` *name* [Command]
: Create a floating window group with the specified name, but do not switch to it.

`gnewbg-dynamic` *name* [Command]
: Create a new dynamic group named NAME in the background.

`gnext` [Command]
: Cycle to the next group in the group list.

`gprev` [Command]
: Cycle to the previous group in the group list.

`gnext-with-window` [Command]
: Cycle to the next group in the group list, taking the current window along.

`gprev-with-window` [Command]
: Cycle to the previous group in the group list, taking the current window along.

`gother` [Command]
: Go back to the last group.

`gmerge` *from* [Command]
: Merge *from* into the current group. *from* is not deleted.

`groups` **&optional** (*fmt* ***group-format***) [Command]
: Display the list of groups with their number and name. **group-format** controls the formatting. The optional argument *fmt* can be used to override the default group formatting.

`vgroups` **&optional** *gfmt wfmt* [Command]
: Like `groups` but also display the windows in each group. The optional arguments *gfmt* and *wfmt* can be used to override the default group formatting and window formatting, respectively.

`gselect` **&optional** *to-group* [Command]
: Accepts numbers to select a group, otherwise grouplist selects.

`gmove` *to-group* [Command]
: Move the current window to the specified group.

`gmove-and-follow` *to-group* [Command]
: Move the current window to the specified group, and switch to it.

`gmove-marked` *to-group* [Command]
: move the marked windows to the specified group.

`gkill` [Command]
: Kill the current group. All windows in the current group are migrated to the next group.

`gkill-other` [Command]
: Kill other groups. All windows in other groups are migrated to the current group.

`grename` *name* [Command]
: Rename the current group.

`grouplist` **&optional** (*fmt* ***group-format***) [Command]
: Allow the user to select a group from a list, like windowlist for groups.

`*list-hidden-groups*` [Variable]
: nil

`*group-top-maps*` [Variable]
: nil

`*default-group-name*` [Variable]
: nil

`add-group` *screen name* **&key** *background* (*type* ***default-group-type***) [Function]
: Create a new group in SCREEN with the supplied name. group names starting with a . are considered hidden groups. Hidden groups are skipped by gprev and gnext and do not show up in the group listings (unless *list-hidden-groups* is T). They also use negative numbers.

`group-add-head` *group head* [Function]
: A head is being added to this group's screen.

`group-add-window` *group window* **&key** *frame raise* **&allow-other-keys** [Function]
: Called when a window is added to the group. All house keeping is already taken care of. Only the group's specific window managing housekeeping need be done. This function accepts keys to inform the group on how to place the window.

`group-button-press` *group button x y child* [Function]
: The user clicked somewhere in the group.

`group-current-head` *group* [Function]
: The group is asked to return its current head.

`group-current-window` *group* [Function]
: The group is asked to return its focused window.

`group-delete-window` *group window* [Function]
: Called when a window is removed from thegroup. All house keeping is already taken care of. Only the group's specific window managing housekeeping need be done.

`group-focus-window` *group win* [Function]
: The group is asked to focus the specified window wherever it is.

`group-indicate-focus` *group* [Function]
: The group is asked to in some way show the user where the keyboard focus is.

`group-lost-focus` *group* [Function]
: The current window was hidden or destroyed or something happened to it. So the group is asked to do something smart about it.

`group-move-request` *group window x y relative-to* [Function]
: The window requested a position change.

`group-raise-request` *group window type* [Function]
: A request has been made to raise the window. TYPE is the type of raise request being made. :MAP means the window has made requested to be mapped. :above means the window has requested to to be placed above its siblings.

`group-remove-head` *group head* [Function]
: A head is being removed from this group's screen.

`group-before-resize-head` *group oh nh* [Function]
: A head is about to be resized on this group's screen.

`group-after-resize-head` *group head* [Function]
: A head has been resized on this group's screen.

`group-resize-request` *group window width height* [Function]
: The window requested a width and/or height change.

`group-root-exposure` *group* [Function]
: The root window got an exposure event. If the group needs to redraw anything on it, this is where it should do it.

`group-startup` *group* [Function]
: Called on all groups while stumpwm is starting up.

`group-suspend` *group* [Function]
: When the group is no longer the current group, this function is called.

`group-sync-all-heads` *group* [Function]
: Called when the head configuration for the group changes.

`group-sync-head` *group head* [Function]

When a head or its usable area is resized, this is called. When the modeline size changes, this is called.

`group-wake-up` *group* [Function]

When the group becomes the current group, this function is called. This call is expected to set the focus.

`really-raise-window` *window* [Function]

Really bring the window to the top of the window stack in group

`*run-or-raise-all-groups*` [Variable]

When this is `T` the `run-or-raise` function searches all groups for a running instance. Set it to NIL to search only the current group.

8.1 Customizing Groups

`*group-formatters*` [Variable]

An alist of characters and formatter functions. The character can be used as a format character in **group-format**. When the character is encountered in the string, the corresponding function is called with a group as an argument. The functions return value is inserted into the string. If the return value isn't a string it is converted to one using `prin1-to-string`.

`*group-format*` [Variable]

The format string that decides what information will show up in the group listing. The following format options are available:

%n	Substitutes the group number translated via *group-number-map*, if there are more windows than *group-number-map* then will use the group-number.
%s	The group's status. Similar to a window's status.
%t	The group's name.

`current-group` **&optional** (*screen* **(current-screen)**) [Function]

Return the current group for the current screen, unless otherwise specified.

9 Screens

StumpWM handles multiple screens.

`snext` [Command]
: Go to the next screen.

`sprev` [Command]
: Go to the previous screen.

`sother` [Command]
: Go to the last screen.

`*run-or-raise-all-screens*` [Variable]
: When this is `T` the `run-or-raise` function searches all screens for a running instance. Set it to `NIL` to search only the current screen. If **run-or-raise-all-groups** is `NIL` this variable has no effect.

9.1 External Monitors

StumpWM refers to each monitor as a head. Heads are logically contained by screens. In a dual-monitor configuration, there will be one screen with two heads. Non-rectangular layouts are supported (frames will not be created in the 'dead zone'.) And message windows will be displayed on the current head–that is, the head to which the currently focused frame belongs.

In addition, StumpWM listens for XRandR events and re-configures the heads to match the new monitor configuration. Occasionally StumpWM will miss an XRandR event, use `refresh-heads` to synchronize the head configuration.

`refresh-heads` **&optional** (*screen* **(current-screen)**) [Command]
: Refresh screens in case a monitor was connected, but a ConfigureNotify event was snarfed by another program.

9.2 Programming With Screens

`current-screen` [Function]
: Return the current screen.

`screen-current-window` *screen* [Function]
: Return the current window on the specified screen

`current-window` [Function]
: Return the current window on the current screen

`*screen-list*` [Variable]
: The list of screens managed by stumpwm.

10 Minor Modes

Like Emacs, StumpWM has the concept of minor modes. These are defined by the macro `DEFINE-MINOR-MODE`. Defining a minor mode creates a class and a set of methods specializing upon it. Minor modes are scoped to a window, head, group, or screen, or they may be unscoped. In addition to this minor modes may be local or global. When a minor mode is global all new instances of the scope object will be created with the minor mode already active in them. Minor modes define their own top level and root level keymaps, as well as hooks that are run upon enabling or disabling the minor mode, and a lighter to display in the mode line.

Minor modes are mixins that get added to the appropriate scope object when enabled. As such minor modes allow the augmenting, modifying, and overriding of default StumpWM behavior by defining methods for the generic functions of the scope object. For example, a minor mode may be scoped to a window and define a method for the generic function `UPDATE-DECORATION` to change how window decoration is handled for the windows it is enabled in.

`define-minor-mode` *mode superclasses slots* **&rest** *options* [Macro]

Define a minor mode as a class to be instantiated when the minor mode is activated. Minor modes are dynamically mixed in to and out of the appropriate object when they are enabled or disabled.

If *SUPERCLASSES* is not provided a default superclass of MINOR-MODE will be provided. *OPTIONS* may include all normal options when defining a class, with the addition of the following options:

- (:SCOPE SCOPE-DESIGNATOR)
  The :SCOPE option determines what object(s) the minor mode can be mixed in with. New scopes can be defined with the macro DEFINE-MINOR-MODE-SCOPE.
- (:GLOBAL (OR T NIL))
  When true the :GLOBAL option changes the way enable methods are defined to track the minor mode and autoenable it in all existing scope objects, as well as autoenabled when new scope objects are instantiated. If the :SCOPE option is :UNSCOPED then this option does not need to be provided.
- (:TOP-MAP spec)
  The minor modes top map is created based upon the provided spec, which must be a list of cons cells whose car is a key sequence and whose cdr is a binding. For example: `(list (cons "C-m x" "echo"))`. This would bind the key sequence `C-m x` to the echo command. A reference to this keymap is stored as a slot in the minor mode object and can be accessed via the reader `MODE-KEYMAP` where `MODE` is the minor mode name.
- (:ROOT-MAP spec)
  The minor modes root map is created based upon the provided spec. The spec is as described in the :TOP-MAP option.
- (:EXPOSE-KEYMAPS (OR T NIL))
  This value is used at macroexpansion time to determine whether or not to generate

keymap variables or store the keymap within the object. When T the variables *MODE-TOP-MAP* and *MODE-ROOT-MAP* will be generated.

- (:REBIND (MEMBER :TOP-MAP :ROOT-MAP :ALL-MAPS))

  This option controls rebinding of the top and root maps. When it is :TOP-MAP the top map is rebound, when it is :ROOT-MAP the root map is rebound, and when it is :ALL-MAPS both the top and root map are rebound. Any rebound map will be rebound to the provided keymap specification. This only has an effect if the minor mode has previously been defined.
- (:LIGHTER T)
  The :LIGHTER option will be used to generate a function returning a string to display in the mode line. When :LIGHTER is NULL a string is generated based upon the mode name. When it is a string that string is used as is. Otherwise :LIGHTER will assumed to be funcallable and used as is. When it is a symbol or a list that doesn't begin with LAMBDA or FUNCTION a warning is issued that DEFINE-MINOR-MODE is assuming it is funcallable. When assumed to be funcallable, it is called with the mode object as its only argument.
- (:LIGHTER-MAKE-CLICKABLE (OR T NIL))
  When :LIGHTER-MAKE-CLICKABLE is T then the :LIGHTER is wrapped in a call to FORMAT-WITH-ON-CLICK-ID, called with the id :ML-ON-CLICK-MINOR-MODE and the mode as a quoted symbol.
- (:LIGHTER-ON-CLICK FUNCTION)
  When :LIGHTER-ON-CLICK is provided it must be a function of arity one, which will be called whenever the minor modes lighter is clicked, with the button code of the click as its only argument. If this is provided then :LIGHTER-MAKE-CLICKABLE is implied to be T.
- (:INTERACTIVE (OR SYMBOL T NIL))
  The :INTERACTIVE option determines whether a command to toggle the minor mode on and off is generated. If it is T then a command with the same name as the minor mode is generated. If it is a symbol then that symbol will be used when defining the command.
- (:ENABLE-WHEN (MODE OBJECT) &BODY BODY)
  When provided, the :ENABLE-WHEN option generates a method for the enable-when generic function. MODE is bound to the mode symbol, and OBJECT is bound to the scope object. If this is not provided, a method is generated which returns T for the minor mode and its scope. If it is provided and is nil, then no method is generated and a method for ENABLE-WHEN which dispatches upon the mode as a symbol and the scope type for the minor mode must be manually defined.
- (:MAKE-HOOKS (OR T NIL))
  When :MAKE-HOOKS is T a set of hook variables are generated. These variables are fourfold: *MODE-HOOK* is run after explicitly enabling the minor mode. *MODE-ENABLE-HOOK* is run when the minor mode is autoenabled. *MODE-DISABLE-HOOK* is run when the minor mode is autodisabled. Finally *MODE-DESTROY-HOOK* is run when the minor mode is explicitly disabled.

- (:DEFINE-COMMAND-DEFINER (OR T NIL))
  When :DEFINE-COMMAND-DEFINER is T a macro is defined for defining commands that are active only when the minor mode is active. Commands defined with this macro have the special variable *MINOR-MODE* bound to the minor mode object in their body. The generated macro is called DEFINE-MODE-COMMAND. This option defaults to T.

Example:

```
(define-minor-mode evil-mode () ()
  (:scope :unscoped)
  (:top-map '(("j" . "move-focus down")
              ("k" . "move-focus up")
              ("h" . "move-focus left")
              ("l" . "move-focus right")
              ("x" . *exchange-window-map*)
              ("C-m b" . "evil-echo")))
  (:lighter "EVIL")
  (:lighter-make-clickable nil))

(define-evil-mode-command evil-echo () ()
  (run-commands "echo"))
```

10.1 Programming With Minor Modes

Minor modes get their power from their ability to override and augment generic functions which are called with the minor mode's scope object as an argument. If you find a function whose behavior you wish to augment or override in the process of writing a minor mode, open an issue or submit a PR to generify the function. Generification is easily done like so:

```
(defun somefun (a b)
  "docstring"
  (otherfun (+ a b)))
;; the above becomes
(defgeneric somefun (a b)
  (:documentation "docstring")
  (:method (a b)
    (otherfun (+ a b))))
```

When defining a minor mode, the programmer may desire to perform setup for the minor mode. This is can be done in three ways. The first is to hang a function upon the minor modes enable hook. However this runs the risk of users potentially clobbering the initialization function, or modifying the hook such that the initialization function is not the first function run. The second way is to define a method for the generic function `update-instance-for-different-class`. This function should specialize upon the minor mode as the second argument. As a final option, one can define before, after, and around methods for the generic function `autoenable-minor-mode` if and only if the method does not access any slots within the object.

One of the pitfalls of minor modes is that they are ultimately enabled by calling `change-class`, which places some restrictions upon where they can be enabled. Specifically, it is implied to be undefined behavior if a minor mode is enabled in an object from within a method which accesses slots of that object. While in practice this has not proven to be an issue at the time of writing, this is undefined behavior and future versions of SBCL may break if this is done.

When writing a minor mode, it is often useful to separate out the desired functionality into its own mixin classes and use those in the superclass list of the minor mode. For example:

```
(define-minor-mode my-mode (my-mixin minor-mode) ())
```

This prevents issues with inheritance and dynamic mixins from cropping up. Since minor modes are just classes, a minor mode can descend from another minor mode. However after enabling the subclass minor mode, the superclass minor mode cannot be enabled. However if the superclass minor mode is enabled first, then the subclass minor mode can be enabled. The easiest way around this is the aforementioned approach of mixins. As an example of the inheritance issue, take the following minor mode definitions:

```
(define-minor-mode x () ())
(define-minor-mode y (x) ())

(enable-minor-mode 'x)
(enable-minor-mode 'y) ; both modes are enabled
;; As opposed to
(enable-minor-mode 'y)
(enable-minor-mode 'x) ; signals an error
```

When enabling and disabling minor modes theres a set of generic functions in charge of determining what object to mix the minor mode in to and whether or not to mix it.

`autoenable-minor-mode` *mode object* [Function]

The core of enabling minor modes within an object. Mixes the minor mode in to the object

Defining a minor mode defines a main method for this generic function which will mix the minor mode into the scope object when called and returns T. This method specializes upon the minor mode symbol and the scope type. The minor mode will only be enabled and the hooks run when the function `ENABLE-WHEN` returns T. Any before after or around methods for this function must not access any slots.

`autodisable-minor-mode` *mode object* [Function]

The core of disabling minor modes within an object. Calls the minor modes on-disable function.

Defining a minor mode defines a main method for this generic function which specializes upon the mode symbol and the mode, and removes the minor mode from the object. Any methods for this function must not access any slots.

`enable-when` *mode object* [Function]

Define methods for this generic function to control when the minor mode should be enabled.

Outside of autoenabling and autodisabling minor modes, there are several generic functions which dispatch upon minor modes and their names.

`minor-mode-global-p` *minor-mode-symbol* [Function]

Return T when MINOR-MODE-SYMBOL denotes a global minor mode

`minor-mode-scope` *minor-mode-symbol* [Function]

Return as a keyword the scope of the minor mode

`minor-mode-enable-hook` *minor-mode-symbol* [Function]

Returns the minor mode enable hook for a given minor mode symbol. This hook is run whenever the minor mode is enabled via autoenable.

`minor-mode-disable-hook` *minor-mode-symbol* [Function]

Returns the minor mode disable hook for a given minor mode symbol. This hook is run whenever the minor mode is disabled via autodisable.

`minor-mode-hook` *minor-mode-symbol* [Function]

Returns the minor mode hook for a given minor mode symbol. This hook is run whenever the minor mode is explicitly enabled.

`minor-mode-keymap` *minor-mode* [Function]

Return the top map for the minor mode

This function has a set of main methods defined which all call the next method to obtain a list of top maps for every minor mode. Any extra keymaps one wishes to add to the minor mode may be added by defining a main method which calls `call-next-method` and returns a flat list. Similarly, an around method may be used which abides by the same rules.

`minor-mode-lighter` *mode* [Function]

Return a string of minor mode lighters.

This function operates similarly to `minor-mode-keymap`, with a main method defined for every minor mode which calls `call-next-method` and returns a flat list. In addition there is a single around method defined which concatenates all these strings together.

There are also a set of regular functions and special variables which which may be of use when working with minor modes.

`enable-minor-mode` *minor-mode* **&optional** *scope-object* [Function]

Enable MINOR-MODE. If MINOR-MODE is global, then enable it in all relevant objects. Otherwise enable it in the current object. If SCOPE-OBJECT is provided, use SCOPE-OBJECT instead of the current object, or include it in the list of current objects if MINOR-MODE is global

`disable-minor-mode` *minor-mode* **&optional** *scope-object* [Function]

Disable MINOR-MODE in the relevant objects.

`list-modes` *object* [Function]

List all minor modes followed by the major mode for OBJECT.

`list-minor-modes` *object* [Function]

List all minor modes active in OBJECT

`current-minor-modes` **&optional** (*screen* **(current-screen)**) [Function]

Return all currently active minor modes.

`minor-mode-enabled-p` *minor-mode* **&optional** (*screen* **(current-screen)**) [Function]

Return T if MINOR-MODE is active

`find-minor-mode` *minor-mode* **&optional** (*screen* **(current-screen)**) [Function]

Return the minor mode object associated with MINOR-MODE.

`*minor-mode*` [Variable]

A dynamic variable bound to the minor mode object when executing a minor mode command.

`*minor-mode-enable-hook*` [Variable]

A hook run whenever a minor mode is enabled. Functions are called with the minor mode symbol and the object they have been added to. This is run when a minor mode is explicitly enabled via enable-minor-mode.

`*minor-mode-disable-hook*` [Variable]

A hook run whenever a minor mode is disabled. Functions are called with the minor mode symbol and the scope object. This is run when a minor mode is explicitly disabled via disable-minor-mode. This is run AFTER the minor mode has been disabled, and is called with the minor mode and the first object it was disabled in.

`*unscoped-minor-modes*` [Variable]

A dynamic variable holding all unscoped minor modes as mixed into the same object.

`minor-mode` [Class]

The root minor mode class. All minor modes are subclasses of this class.
Direct Superclasses: standard-object
Direct Subclasses:
Direct Slots:

10.2 Minor Mode Scopes

Minor modes can be scoped to different objects in a rather arbitrary manner. These scopes are defined by the macro `DEFINE-MINOR-MODE-SCOPE`. Because minor modes are implemented as mixins, the object returned by a scopes current object function must be a class instance.

`define-minor-mode-scope` (*designator class* **&optional** *filter-type*) **&body** *retrieve-current-object* [Macro]

Define a minor mode scope for use with DEFINE-MINOR-MODE. This generates a call to ADD-MINOR-MODE-SCOPE which is evaluated when compiled, loaded, or executed. DESIGNATOR should be a keyword and TYPE should denote a class, while FILTER-TYPE should denote a general type. RETRIEVE-CURRENT-OBJECT should be a thunk body which returns the current object for this scope.

`define-descended-minor-mode-scope` *designator parent* **&key** *class filter-type retrieve-current-object* [Macro]

Define a descended scope which inherits the parents type and functions unless provided.

`add-minor-mode-scope` *designator type current-object-thunk* **&optional** *filter-type* [Function]

Add a list of the TYPE, CURRENT-OBJECT-THUNK, and ALL-OBJECTS-THUNK, under DESIGNATOR in the minor mode scope hash table.

When a minor mode is defined its scope is looked up and validated by the function `VALIDATE-SCOPE`. This function takes a scope and a list of superclasses, and ensures that the scope can descend from the superclasses scopes. This restricts the valid scopes to ensure that a minor mode scoped to `:GROUP` cant be a subclass of a minor mode scoped to `:WINDOW`, for example. However there is a way to override this by explicitly stating that two otherwise incompatible scopes are compatible. This is done by defining methods for the generic function `VALIDATE-SUPERSCOPE` which dispatch upon the scope designators. Such methods should return at least one value, indicating if the superscope is a valid parent of the scope. If multiple values are returned, the second value must indicate whether the superscope is an invalid parent of the scope. For example:

```
(defmethod stumpwm:validate-superscope ((c (eql :id1)) (p (eql :id2)))
  "Explicitly allow id1 to descend from id2"
  (values t nil))

(defmethod stumpwm:validate-superscope ((c (eql :id3)) (p (eql :id4)))
  "Explicitly prevent id3 from descending from id4"
  (values nil t))
```

When defining and using scopes the type specifier is important; it is used to determine what minor modes should be mixed into an object when it is created. For this reason it is important when defining a minor mode or minor mode scope to understand the type hierarchy. It may also be in the programmers best interests to define an accompanying type.

The following scopes are predefined:

- Designator: `:UNSCOPED`, type: `T` Current object: return the global unscoped object.
- Designator: `:SCREEN`, type: `SCREEN` Current object: return the current screen.
- Designator: `:GROUP`, type: `GROUP` Current object: return the current group.
- Designator: `:TILE-GROUP`, type: `TILE-GROUP` Current object: return the current group.
- Designator: `:DYNAMIC-GROUP`, type: `DYNAMIC-GROUP` Current object: return the current group.
- Designator: `:FLOAT-GROUP`, type: `FLOAT-GROUP` Current object: return the current group.

- Designator: `:TILING-NON-DYNAMIC-GROUP`, type: `TILE-GROUP` Current object: return the current group when it is a non-dynamic tiling group.

- Designator: `:HEAD`, type: `HEAD` Current object: return the current head.

- Designator: `:FRAME`, type: `FRAME` Current object: return the current frame when in a tiling group.

- Designator: `:FRAME-EXCLUDING-HEAD`, type: `ONLY-FRAME-NO-HEADS` Descends from minor mode scope `:FRAME`.
- Designator: `:WINDOW`, type: `WINDOW`
  Current object: return the current window.
  All objects: collect every window from every group in the current screen.
- Designator: `:TILE-WINDOW`, type: `TILE-WINDOW`
  Current object: return the current window.
  All objects: collect every window from every tiling group, filtering all floating windows.
- Designator: `:FLOAT-WINDOW`, type: `FLOAT-WINDOW`
  Current object: return the current window.
  All objects: colelct every window from every group, filtering all non floating windows.

10.3 Minor Mode Example

A simple example of a minor mode is a version of the Emacs modes viper or evil for StumpWM. Such a minor mode might look like this:

```
(define-minor-mode swm-evil-mode () ()
  (:scope :screen)
  (:interactive t)
  (:top-map '(("i" . "swm-evil-mode")
              ("j" . "move-focus down")
              ("k" . "move-focus up")
              ("h" . "move-focus left")
              ("l" . "move-focus right")
              ("p" . "pull-hidden-previous")
              ("n" . "pull-hidden-next")
              ("S" . "hsplit")
              ("s" . "vsplit")
              ("r" . "remove-split")
              ("g" . *groups-map*)
              ("x" . *exchange-window-map*)))
  (:lighter-make-clickable nil)
  (:lighter "EVIL"))
```

In the above example, the minor mode `swm-evil-mode` is defined, alongside a command of the same name which toggles it on and off. The minor mode is scoped to a screen, meaning that upon activation it will be dynamically mixed in to the screen object. The lighter is the string `"EVIL"` and the lighter is not made clickable.

When defining a minor mode top map it is important to avoid multi-key bindings that clobber the prefix key. For example, if the prefix key is `C-t` then defining the keybinding `C-t n` in the top map of a minor mode is an error. Instead bind the key `n` in the minor mode's root map.

As another example we can define a frame topbar mode. This should adjust every frame to leave extra space at the top of the frame to display a bar of some sort. The following assumes that the functions `frame-display-height` and `frame-display-y` are generic.

```
(defclass frame-topbar ()
  ((frame-topbar-height :initform 10 :accessor frame-topbar-height)))

(defmethod frame-display-height :around (group (frame frame-topbar))
  (let ((height (call-next-method)))
    (- height (frame-topbar-height frame))))

(defmethod frame-display-y :around (group (frame frame-topbar))
  (let ((y (call-next-method)))
    (+ y (frame-topbar-height frame))))

(define-minor-mode frame-bar (frame-topbar minor-mode) ()
  (:global t)
  (:scope :frame)
  (:lighter "T-BAR")
  (:interactive frame-topbar-mode))

(defmethod update-instance-for-different-class :after
    (prev (obj frame-bar) &rest rest)
  (declare (ignore prev rest))
  (when (frame-window obj)
    (let* ((group (window-group (frame-window obj)))
           (windows (frame-windows group obj)))
      (mapc #'maximize-window windows))))
```

In the above example, a class is defined which holds the height of the frame topbar. Then two around methods are defined such that windows querying the frame for their y position and height get an updated value reflecting the topbars presence. Then a minor mode is defined which inherits from the class we defined. It is scoped to frames and is a global minor mode, so it will be enabled in all existing frames and any other frames as they are created. Finally the initialization is handled in the after method for update-instance-for-different-class, which updates every window to have a new size which respects the topbar.

The implementation of the actual topbar is left as an exercise for the reader.

11 Internals

11.1 IO Loop

StumpWM's internal loop is implemented by a generic multiplexing I/O loop for listening to I/O events from multiple sources. The model is as follows:

An I/O multiplexer is represented as an object, with which I/O channels can be registered to be monitored for events when the I/O loop runs. An I/O channel is any object for which the generic functions IO-CHANNEL-IOPORT, IO-CHANNEL-EVENTS and IO-CHANNEL-HANDLE are implemented.

IO-CHANNEL-IOPORT, given an I/O multiplexer and an I/O channel, should return the underlying system I/O facility that the channel operates on. The actual objects used to represent an I/O facility depends on the Lisp implementation, operating system and the specific I/O loop implementation, but, for example, on Unix implementations they will likely be numeric file descriptors. The I/O loop implementation implements IO-CHANNEL-IOPORT methods for the facilities it understands (such as FD-STREAMs on SBCL), so user-implemented channels should simply call IO-CHANNEL-IOPORT recursively on whatever it operates on.

IO-CHANNEL-EVENTS, given an I/O channel, should return a list of the events that the channel is interested in. See the documentation for IO-CHANNEL-EVENTS for further details.

The I/O loop guarantees that it will check what events a channel is interested in when it is first registered, and also at any time the channel has been notified of an event. If the channel changes its mind at any other point in time, it should use the IO-LOOP-UPDATE function to notify the I/O loop of such changes. The I/O loop may very well also update spuriously at other times, but such updates are not guaranteed.

IO-CHANNEL-HANDLE is called by the I/O loop to notify a channel of an event.

An I/O multiplexer is created with a MAKE-INSTANCE call on the class of the desired multiplexer implementation. If the code using the multiplexer has no certain preferences on an implementation (which should be the usual case), the variable *DEFAULT-IO-LOOP* points to a class that should be generally optimal given the current Lisp implementation and operating system.

Given a multiplexer, channels can be registered with it using IO-LOOP-ADD, unregistered with IO-LOOP-REMOVE, and updated with IO-LOOP-UPDATE (as described above). Call IO-LOOP on the multiplexer to actually run it.

`*default-io-loop*` [Variable]

> The default I/O loop implementation. Should be generically optimal for the given LISP implementation and operating system.

`*current-io-loop*` [Variable]

> Dynamically bound to the I/O loop currently running, providing an easy way for event callbacks to register new channels.

`*current-io-channel*` [Variable]

> While processing an I/O channel, this variable is dynamically bound to the channel in question. This is provided primarily for error-handling code.

`io-channel-ioport` *io-loop channel* [Function]

Returns the I/O facility operated on by CHANNEL, in a representation understood by IO-LOOP. CHANNEL may be either an I/O channel or an object representing an underlying I/O facility, such as a stream object. An I/O loop implementation should implement methods for any primitive I/O facilities that it can monitor for events, and abstract channels should return whatever IO-CHANNEL-IOPORT returns for the primitive facility that it operates on.

An I/O channel may also return NIL to indicate that it is only interested in purely virtual events, such as :TIMEOUT or :LOOP.

`io-channel-events` *channel* [Function]

Returns a list of events that CHANNEL is interested in. An event specification may be a simple symbol, or a list of a symbol and additional data for the event. Specific I/O loop implementations may implement additional events, but the following event specifications should be supported by all I/O loops:

:READ – The channel will be notified when its I/O port can be read from without blocking.

:WRITE – The channel will be notified when its I/O port can be written to without blocking.

(:TIMEOUT TIME-SPEC) – TIME-SPEC is a point in time in the same units as from (GET-INTERNAL-REAL-TIME), at which point the channel will be notified. It is permissible for TIME-SPEC to be a real number of any representation, but the system does not guarantee any particular level of accuracy.

:LOOP – The channel will be notifed for each iteration of the I/O loop, just before blocking for incoming events. This should be considered a hack to be avoided, but may be useful for certain libraries (such as XLIB).

If, at any time, an empty list is returned, the channel is unregistered with the I/O loop.

The I/O loop will check what events a channel is interested in when it is first registered with the loop, and whenever the channel has been notified of an event. If the channel changes its mind at any other point in time, it should use the IO-LOOP-UPDATE function to notify the I/O loop of such changes. The I/O loop may also update spuriously at any time, but such updates are not guaranteed.

`io-channel-handle` *channel event* **&key &allow-other-keys** [Function]

Called by the I/O loop to notify a channel that an event has occurred. EVENT is the symbol corresponding to the event specification from IO-CHANNEL-EVENTS (that is, :READ, :WRITE, :TIMEOUT or :LOOP). A number of keyword arguments with additional data specific to a certain event may also be passed, but no such arguments are currently defined.

`io-loop-add` *io-loop channel* [Function]

Add a channel to the given I/O multiplexer to be monitored.

`io-loop-remove` *io-loop channel* [Function]

Unregister a channel from the I/O multiplexer.

`io-loop-update` *io-loop channel* [Function]

Make the I/O loop update its knowledge of what events CHANNEL is interested in. See the documentation for IO-CHANNEL-EVENTS for more information.

`io-loop` *io-loop* **&key** *description* **&allow-other-keys** [Function]

Run the given I/O multiplexer, watching for events on any channels registered with it. IO-LOOP will return when it has no channels left registered with it.

11.2 Internal Functions Documentation

`*executing-stumpwm-command*` [Variable]

True when executing external commands.

`*suppress-abort-messages*` [Variable]

Suppress abort message when non-nil.

`refresh-time-zone` [Command]

Refresh the time zone information from the system.

If you change the system time zone while StumpWM is running you can run this command to make StumpWM notice the change.

`getsel` [Command]

Echo the X selection.

`putsel` *string* [Command]

Stuff the string *string* into the x selection.

`copy-unhandled-error` [Command]

When an unhandled error occurs, StumpWM restarts and attempts to continue. Unhandled errors should be reported to the mailing list so they can be fixed. Use this command to copy the unhandled error and backtrace to the X11 selection so you can paste in your email when submitting the bug report.

`define-stumpwm-command` **&rest** *args* [Function]

nil

`set-contrib-dir` [Command]

Deprecated, use ‘add-to-load-path’ instead

12 Interacting With Unix

`run-shell-command` *cmd* **&optional** *collect-output-p* [Command]

Run the specified shell command. If *collect-output-p* is `T` then run the command synchonously and collect the output. Be careful. If the shell command doesn't return, it will hang StumpWM. In such a case, kill the shell command to resume StumpWM.

`programs-in-path` **&optional** *full-path* (*path* **(split-string (getenv PATH) :)**) [Function]

Return a list of programs in the path. If *full-path* is *t* then return the full path, otherwise just return the filename. *path* is by default the `PATH` evironment variable but can be specified. It should be a string containing each directory seperated by a colon.

`pathname-is-executable-p` *pathname* [Function]

Return T if the pathname describes an executable file.

`pathname-as-directory` *pathspec* [Function]

Converts the non-wild pathname designator PATHSPEC to directory form.

`run-or-raise` *cmd props* **&optional** (*all-groups* ***run-or-raise-all-groups***) (*all-screens* ***run-or-raise-all-screens***) [Function]

Run the shell command, *cmd*, unless an existing window matches *props*. *props* is a property list with the following keys:

`:class`	Match the window's class.
`:instance`	Match the window's instance or resource-name.
`:role`	Match the window's `WM_WINDOW_ROLE`.
`:title`	Match the window's title.

By default, the global **run-or-raise-all-groups** decides whether to search all groups or the current one for a running instance. *all-groups* overrides this default. Similarily for **run-or-raise-all-screens** and *all-screens*.

`*shell-program*` [Variable]

The shell program used by `run-shell-command`.

`getenv` *var* [Function]

Return the value of the environment variable.

`(setf getenv)` *val var* [Function]

Set the value of the environment variable, *var* to *val*.

13 Interacting With X11

`set-x-selection` *text* **&optional** (*selection* ***default-selections***) [Function]

Set the X11 selection string to *string*.

`get-x-selection` **&optional** *timeout* (*selection* ***default-selections***) [Function]

Return the x selection no matter which client owns it.

`*default-selections*` [Variable]

A keyword or list, one of: :primary or '(:primary) uses only the "primary" selection :clipboard or '(:clipboard) uses only the "clipboard" selection Both can be specified in a list like '(:primary :clipboard). In this case, set-x-selection will clobber both, and get-x-selection will default to the first item.

`*x-selection*` [Variable]

This is a plist of stumpwm's current selections. The different properties are generally set when killing text in the input bar.

14 Miscellaneous Commands

The following is a list of commands that don't really fit in any other

`split-string` *string* **&optional** (*separators* [Function]

) Splits STRING into substrings where there are matches for SEPARATORS. Each match for SEPARATORS is a splitting point. The substrings between the splitting points are made into a list which is returned. ***If SEPARATORS is absent, it defaults to `"[ ftnrv]+"`.

If there is match for SEPARATORS at the beginning of STRING, we do not include a null substring for that. Likewise, if there is a match at the end of STRING, we don't include a null substring for that.

Modifies the match data; use 'save-match-data' if necessary.

`argument-line-end-p` *input* [Function]

Return T if we're outta arguments from the input line.

`argument-pop` *input* [Function]

Pop the next argument off.

`argument-pop-rest` *input* [Function]

Return the remainder of the argument text.

`*display*` [Variable]

The display for the X server

`input-delete-region` *input start end* [Function]

Delete the region between start and end in the input string

`input-goto-char` *input point* [Function]

Move the cursor to the specified point in the string

`input-point` *input* [Function]

Return the position of the cursor.

`input-substring` *input start end* [Function]

Return a the substring in INPUT bounded by START and END.

`input-validate-region` *input start end* [Function]

Return a value pair of numbers where the first number is `<` the second and neither excedes the bounds of the input string.

`list-directory` *dirname* [Function]

Returns a fresh list of pathnames corresponding to the truenames of all files within the directory named by the non-wild pathname designator DIRNAME. The pathnames of sub-directories are returned in directory form - see PATHNAME-AS-DIRECTORY.

`move-to-head` *list elt* [Macro]

Move the specified element in in LIST to the head of the list.

`no-focus` *group last-win* [Function]
: don't focus any window but still read keyboard events.

`*record-last-msg-override*` [Variable]
: assign this to T and messages won't be recorded. It is recommended this is assigned using LET.

`*toplevel-io*` [Variable]
: Top-level I/O loop

14.1 Menus

14.2 Menus

There are three different types of menus in StumpWM; single selection menus; interactive menus; and batch menus. Single-selection menus, as the name suggests, are used to pick a single item from a list. Interactive menus are used for marking multiple selections. Batch menus are used for performing actions on multiple menu items. Both batch and interactive menus share the same navigational keybindings, which are found in the table below. These can be customized by modifying the **menu-map** variable. Commands specific to each menu type can be modified by **single-menu-map** and **batch-menu-map**.

`C-p`
`Up`
`k`
	Highlight the previous menu option.

`C-n`
`Down`
`j`
	Highlight the next menu option.

`S-Down`	Scroll the entire page down one entry.
`S-Up`	Scroll the entire page up one entry.
`PageUp`	Scroll up one page.
`PageDown`	Scroll down one page.
`C-g` `ESC`	Abort the menu.

In addition, you can customize the number of items shown at a time (a page) with the **menu-maximum-height** variable. The default value, `50`, limits the menu size to 50 items. Setting it to `nil` will remove the limit on how many menu entries are shown (be careful, this can crash X11 when attempting to display a large amount of items).

`*menu-map*` [Variable]
: The keymap used by the interactive menu.

`menu-page-up` *menu* [Function]
: Move a whole page down in the menu

`menu-page-down` *menu* [Function]
: Move a whole page up in the menu

`menu-up` *menu* [Function]
: Move menu cursor up

`menu-down` *menu* [Function]
: Move menu cursor down

`menu-scroll-up` *menu* [Function]
: Scroll the menu up

`menu-scroll-down` *menu* [Function]
: Scroll the menu down

`menu-abort` *menu* [Function]
: What to do when exiting the menu without results. Must signal :menu-quit with the result.

`menu-backspace` *menu* [Function]
: What occurs when backspace is pressed in a menu

`menu-entry-apply` *menu-entry function* [Function]
: Apply FUNCTION to the data portion of the menu entry.

`menu-entry-display` *menu-entry* [Function]
: Generates a string suitable for displaying in a menu

`menu-finish` *menu* [Function]
: What to do when exiting the menu with results. Must signal :menu-quit with the result.

`command-menu` *screen items command-list* **&key** (*prompt* **Select:**) (*initial-selection* **0**) *extra-keymap* [Function]
: Use batch-menu to make selections and run commands specified in command-list.

SCREEN: The screen to display the menu on.

ITEMS: The items to be shown in the list. This is expected to be a list of `menu-item`s.

COMMAND-LIST: A list of entries defining the commands associated with each mark. Only marks that are defined are allowed in the menu. The format for these entries is (mark-character function calling-options).

Available calling-options: :single (Default) Each value is passed separately to the supplied function. :all all values selected with this mark are passed to the function in a list.

Example: '((#d 'delete-window) (#m 'move-multiple-windows :all))

14.2.1 Single Selection Menus

Single selection menus can be searched; start typing when the menu is active, and the results are immediately filtered. Use `RET` to selected the highlighted option.

`RET` Select the highlighted option.

`*single-menu-map*` [Variable]
: The keymap used by single selection menus in addition to *menu-map*

`select-from-menu` *screen table* **&optional** (*prompt* **Search:**) (*initial-selection* **0**) *extra-keymap* (*filter-pred* **(function menu-item-matches-regexp)**) [Function]

Prompt the user to select from a menu on SCREEN. TABLE can be a list of values or a nested list. If it's a nested list, the first element in the sublist is displayed in the menu. What is displayed as menu items must be strings.

EXTRA-KEYMAP can be a keymap whose bindings will take precedence over the default bindings.

FILTER-PRED should be a a function returning T when a certain menu item should be visible to the user. It should accept arguments

ITEM-STRING (the string shown to the user), ITEM-OBJECT (the object corresponding to the menu item), and USER-INPUT (the current user input). The default is MENU-ITEM-MATCHES-REGEXP. Returns the selected element in TABLE or nil if aborted.

14.2.2 Batch Menus

Batch menus provide a menu that allows the user to mark items. Items are marked by highlighting an item, then pressing a corresponding key. The key pressed depends on the menu being shown, and the desired action. For example, in a menu allowing users to manage windows, windows to be closed/removed could be marked by `d`, and windows to be raised could be marked by `r`. All available actions and their keybindings are shown below. These can be customized with **batch-menu-map**.

`n` `Space`	Highlight the next item.
`p`	Highlight the previous item.
`u`	Unmark the selected item, then move the cursor down.
`DEL`	Unmark the selected item, then move the cursor up if it is not at the top of the menu.
`x` `RET`	Exit the menu and perform the actions associated with each mark.

`*batch-menu-map*` [Variable]

The keymap used by batch-menu menus in addition to *menu-map*

`select-from-batch-menu` *screen table* **&key** (*prompt* **Select:**) *allowed-markers* (*initial-selection* **0**) *extra-keymap* [Function]

Prompt the user with a menu that allows them to mark each item with a character. They can exit the menu by pressing enter, or whatever key is mapped to 'menu-finish' in *menu-map*. Value returned is an alist, where the cdr of each entry is a list of items that were marked with that character. Note that the lisp printer cannot distinguish between '(a . (b c d)) and '(a b c d).

Example when "foo" and "bar" are marked with '#d', and "baz" is not marked: ((#d "foo" "bar") (NIL "baz")) ALLOWED-MARKERS is a list of characters. If this parameter is specified, no other markers are allowed. EXTRA-KEYMAP can be a keymap whose bindings will take precedence over the default bindings.

14.3 StumpWM's Data Directory

If you want to store StumpWM data between sessions, the recommended method is to store them in `~/.stumpwm.d/`. StumpWM supplies some functions to make doing this easier.

`*data-dir*` [Variable]

The directory used by stumpwm to store data between sessions.

`data-dir-file` *name* **&optional** *type* [Function]

Return a pathname inside stumpwm's data dir with the specified name and type

`with-data-file` (*s file* **&rest** *keys* **&key** *(if-exists supersede)* **&allow-other-keys**) **&body** *body* [Macro]

Open a file in StumpWM's data directory. keyword arguments are sent directly to OPEN. Note that IF-EXISTS defaults to :supersede, instead of :error.

14.4 Debugging StumpWM

`*debug-level*` [Variable]

Set this variable to a number `>` 0 to turn on debugging. The greater the number the more debugging output.

`*debug-stream*` [Variable]

This is the stream debugging output is sent to. It defaults to *error-output*. It may be more convenient for you to pipe debugging output directly to a file.

`*debug-expose-events*` [Variable]

Set this variable for a visual indication of expose events on internal StumpWM windows.

`redirect-all-output` *file* [Function]

Elect to redirect all output to the specified file. For instance, if you want everything to go to ~/.stumpwm.d/debug-output.txt you would do:

```
(redirect-all-output (data-dir-file "debug-output" "txt"))
```

14.5 Sending a Bug Report

While StumpWM's code-base is quite mature, it still contains some bugs. If you encounter one here are some guidelines for making sure the developers can fix it:

- Include a procedure for reproducing the bug/bad behavior. Ideally this will include numbered steps starting with instructions on how you start StumpWM. Also include what the expected behavior was.
- Be as detailed as possible. Then add more detail!
- Make sure its not something you introduced by using an empty `.xinitrc` containing only '`exec /path/to/stumpwm`'.
- Make sure the bug is present even when `.stumpwmrc` is empty.
- If you are using the git version, include the hash of the master branch, or better include the commit when you started to notice the bug.

- If you have code that fixes the bug, then open a pull request at `https://github.com/stumpwm/stumpwm/compare/`.
- If you don't have code to fix the bug, then open an issue at `https://github.com/stumpwm/stumpwm/issues/new`.

14.6 Timers

StumpWM has a timer system similar to that of *Emacs*.

`idle-time` *screen* [Function]
: Returns the time in seconds since idle according to the root window of the 'screen'.

`run-with-timer` *secs repeat function* **&rest** *args* [Function]
: Perform an action after a delay of SECS seconds. Repeat the action every REPEAT seconds, if repeat is non-nil. SECS and REPEAT may be reals. The action is to call FUNCTION with arguments ARGS.

`cancel-timer` *timer* [Function]
: Remove TIMER from the list of active timers.

`timer-p` *timer* [Function]
: Return T if TIMER is a timer structure.

14.7 Getting Help

`describe-key` *keys* [Command]
: Either interactively type the key sequence or supply it as text. This command prints the command bound to the specified key sequence.

`describe-variable` *var* [Command]
: Print the online help associated with the specified variable.

`describe-function` *fn* [Command]
: Print the online help associated with the specified function.

`describe-command` *com* [Command]
: Print the online help associated with the specified command.

`where-is` *cmd* [Command]
: Print the key sequences bound to the specified command.

`modifiers` [Command]
: List the modifiers stumpwm recognizes and what MOD-X it thinks they're on.

`which-key-mode` [Command]
: Toggle which-key-mode

`lookup-command` *keymap command* [Function]
: Return a list of keys that are bound to command

`*help-map*` [Variable]
: Help related bindings hang from this keymap

`*help-keys*` [Variable]

The list of keys used to invoke the help command.

15 Colors

When specifying a color, it is possible to use its X11 Color Name (usually in the file `/etc/X11/rgb.txt`). You can also use a six digit hex string prefixed by a '#' character in the same way that you can specify colors in HTML.

All text printed by StumpWM is run through a coloring engine before being displayed. All color commands start with a '`^`' (caret) character and apply to all text after it.

`^0-9` A caret followed by a single digit number changes the foreground color to the specified color. A '`*`' can be used to specify the normal color. See the color listing below.

`^0-90-9` A caret followed by two digits sets the foreground and background color. The first digit refers to the foreground color and the second digit to the background color. A '`*`' can be used in place of either digit to specify the normal color. See the color listing below.

`^B` Turn on bright colors.

`^b` Turn off bright colors.

`^n` Use the normal background and foreground color.

`^R` Reverse the foreground and background colors.

`^r` Turn off reverse colors.

`^[` Push the current colors onto the color stack. The current colors remain unchanged.

`^]` Pop the colors off the color stack.

`^>` Align the rest of the line to the right of the window.

`^f<n>` Sets the current font to the font at index n in the screen's font list.

`^(<modifier> &rest arguments)`
Allows for more complicated color settings: <modifier> can be one of :fg, :bg, :reverse, :bright, :push, :pop, :font and :>. The arguments for each modifier differ:

- :fg and :bg take a color as an argument, which can either be a numeric index into the color map or a hexadecimal color in the form of "#fff" or "#ffffff".
- :reverse and :bright take either t or nil as an argument. T enables the setting and nil disables it.
- :push and :pop take no arguments. :push pushes the current settings onto the color stack, leaving the current settings intact. :pop pops color settings off the stack, updating the current settings.
- :font takes an integer that represents an index into the screen's list of fonts, or, possibly, a literal font object that can immediately be used. In a string you'll probably only want to specify an integer.
- :> takes no arguments. It triggers right-alignment for the rest of the line.

- :on-click takes one or more arguments and registers the following text area as clickable. The initial argument must be an ID registered using the function REGISTER-ML-ON-CLICK-ID.
- :on-click-end takes no arguments. It marks the end of clickable text begun by :on-click.

`^^` Print a regular caret.

The default colors are made to resemble the 16 VGA colors and are:

0 black

1 red

2 green

3 yellow

4 blue

5 magenta

6 cyan

7 white

There are only 8 colors by default but 10 available digits. The last two digits are left up to the user. Section 15.1 [Behind The Scenes Look At Colors], page 90, for information on customizing colors.

15.1 Behind The Scenes Look At Colors

Color indexes are stored in **colors** as a list. The default list of colors leave 2 slots for the user to choose. If you'd like to use '`Papaya Whip`' and '`Dark Golden Rod 3`' you might eval the following:

```
(setf *colors* (append *colors*
                       (list "PapayaWhip"
                             "DarkGoldenRod3")))
(update-color-map (current-screen))
```

Of course, you can change all the colors if you like.

Additionally, both the normal and bright versions of a color can be specified by using a list of the form `(normal-color bright-color)`, for instance:

```
(setf *colors* (append *colors*
                       (list (list "PeachPuff" "PapayaWhip")
                             (list "DarkGoldenRod3" "PaleGoldenrod"))))
(update-color-map (current-screen))
```

`parse-color-string` *string* [Function]

Parse a color-coded string into a list of strings and color modifiers

`uncolorify` *string* [Function]

Remove any color markup in STRING

`*colors*` [Variable]

Eight colors by default. You can redefine these to whatever you like and then call (update-color-map).

`update-color-map` *screen* [Function]

Read *colors* and cache their pixel colors for use when rendering colored text.

16 Hooks

StumpWM exports a number of hooks you can use to add customizations; like hooks in Emacs, you add to a hook with the `add-hook` function. For example:

```
(stumpwm:add-hook 'stumpwm:*new-window-hook* 'my-new-window-custos)
```

adds your `my-new-window-custos` function to the list of functions called when a new window appears.

`add-hook` *hook fn* [Macro]

Add *function* to the *hook-variable*. For example, to display a message whenever you switch frames:

```
(defun my-rad-fn (to-frame from-frame)
  (stumpwm:message "Mustard!"))

(stumpwm:add-hook stumpwm:*focus-frame-hook* 'my-rad-fn)
```

`remove-hook` *hook fn* [Macro]

Remove the specified function from the hook.

`remove-all-hooks` *hook* [Macro]

Remove all functions from a hook

`run-hook` *hook* [Function]

Call each function in HOOK.

`run-hook-with-args` *hook* **&rest** *args* [Function]

Call each function in HOOK and pass args to it.

The following hooks are available:

`*new-window-hook*` [Hook]

A hook called whenever a window is added to the window list. This includes a genuinely new window as well as bringing a withdrawn window back into the window list.

`*destroy-window-hook*` [Hook]

A hook called whenever a window is destroyed or withdrawn.

`*focus-window-hook*` [Hook]

A hook called when a window is given focus. It is called with 2 arguments: the current window and the last window (could be nil).

`*place-window-hook*` [Hook]

A hook called whenever a window is placed by rule. Arguments are window group and frame

`*start-hook*` [Hook]

A hook called when stumpwm starts.

`*internal-loop-hook*` [Hook]

A hook called inside stumpwm's inner loop.

`*focus-frame-hook*` [Hook]
: A hook called when a frame is given focus. The hook functions are called with 2 arguments: the current frame and the last frame.

`*new-frame-hook*` [Hook]
: A hook called when a new frame is created. The hook is called with the frame as an argument.

`*message-hook*` [Hook]
: A hook called whenever stumpwm displays a message. The hook function is passed any number of arguments. Each argument is a line of text.

`*top-level-error-hook*` [Hook]
: Called when a top level error occurs. Note that this hook is run before the error is dealt with according to *top-level-error-action*.

`*focus-group-hook*` [Hook]
: A hook called whenever stumpwm switches groups. It is called with 2 arguments: the current group and the last group.

`*hooks-enabled-p*` [Hook]
: Controls whether hooks will actually run or not

`*remove-split-hook*` [Hook]
: A hook called when a split is removed. the hook is called with the current frame and removed frame as arguments.

`*key-press-hook*` [Hook]
: A hook called whenever a key under *top-map* is pressed. It is called with 3 argument: the key, the (possibly incomplete) key sequence it is a part of, and command value bound to the key.

`*root-click-hook*` [Hook]
: A hook called whenever there is a mouse click on the root window. Called with 4 arguments, the screen containing the root window, the button clicked, and the x and y of the pointer.

`*click-hook*` [Hook]
: A hook called whenever there is a mouse click. Called with 4 arguments, the screen containing the window (or nil if there isn't one), the button clicked, and the x and y of the pointer.

`*mode-line-click-hook*` [Hook]
: Called whenever the mode-line is clicked. It is called with 4 arguments, the mode-line, the button clicked, and the x and y of the pointer.

`*urgent-window-hook*` [Hook]
: A hook called whenever a window sets the property indicating that it demands the user's attention

`*event-processing-hook*` [Hook]
: A hook called inside stumpwm's inner loop, before the default event processing takes place. This hook is run inside (with-event-queue ...).

`*pre-command-hook*` [Hook]
: Called before a command is called. It is called with 1 argument: the command as a symbol.

`*post-command-hook*` [Hook]
: Called after a command is called. It is called with 1 argument: the command as a symbol.

`*menu-selection-hook*` [Hook]
: Called after an item is selected in the windows menu. It is called with 1 argument: the menu.

`*new-head-hook*` [Hook]
: A hook called whenever a head is added. It is called with 2 arguments: the new head and the current screen.

`*command-mode-end-hook*` [Hook]
: A hook called whenever command mode is ended

`*command-mode-start-hook*` [Hook]
: A hook called whenever command mode is started

`*destroy-mode-line-hook*` [Hook]
: Called whenever the mode-line is destroyed. It is called with argument, the mode-line

`*quit-hook*` [Hook]
: A hook called when stumpwm quits.

`*restart-hook*` [Hook]
: A hook called when stumpwm restarts.

`*selection-notify-hook*` [Hook]
: Called after a :selection-notify event is processed. It is called with 1 argument: the selection as a string.

`*split-frame-hook*` [Hook]
: A hook called when a frame is split. the hook is called with the old frame (window is removed), and two new frames as arguments.

17 Modules

A module is an ASDF system that adds additional functionality to StumpWM. StumpWM searches for modules in the **data-dir**`/modules` directory. By default this is `~/.stumpwm.d/modules`.

Officially supported modules exist in a separate repository within the StumpWM organization on github. You can install the latest copy by issuing `make install-modules` from StumpWM's root source directory. This will run:

```
git clone git@github.com:stumpwm/stumpwm-contrib.git ~/.stumpwm.d/modules
```

`load-module` *name* [Command]
: Loads the contributed module with the given NAME.

`list-modules` [Function]
: Return a list of the available modules.

`*load-path*` [Variable]
: A list of paths in which modules can be found, by default it is populated by any asdf systems found in '*module-dir*' set from the configure script when StumpWM was built, or later by the user using 'add-to-load-path'

`add-to-load-path` *path* [Command]
: If 'PATH' is not in '*LOAD-PATH*' add it, check if 'PATH' contains an asdf system, and if so add it to the central registry

`init-load-path` *path* [Function]
: Recursively builds a list of paths that contain modules, then add them to the load path. This is called each time StumpWM starts with the argument '*module-dir*'

`find-module` *name* [Function]
: nil

17.1 Writing Modules

Make sure to read Chapter 18 [Hacking], page 99. If you are familiar with writing lisp packages for ASDF then you can jump in and get started. In either case, quicklisp ships a `quickproject` package that makes setting up a new module very easy. After installing quicklisp (see the README.md for a link):

We're going to put our new module in the `modules/` directory of **data-dir** so that it will be immediately loadable by StumpWM.

First make the directory `new-module`, then from a REPL issue:

```
(ql:quickload "quickproject")
(quickproject:make-project #p"~/.stumpwm.d/modules/new-module" :depends-on '(stumpwm)
```

This will create:

```
  -rw-rw-r--  1 dave dave   68 Apr  6 19:38 package.lisp
  -rw-rw-r--  1 dave dave   53 Mar 16  2014 README.txt
  -rw-rw-r--  1 dave dave  271 Mar 16  2014 new-module.asd
```

```
-rw-rw-r--  1 dave dave 1.8K Apr  6 17:51 new-module.lisp
```

The file `new-module.lisp` will contain the actual implementation of your module. ASDF requires two other files in order to understand how to load and compile your module. They are `new-module.asd` and `package.lisp`. In our example, `new-module.asd` should contain:

```
(asdf:defsystem #:new-module
  :serial t
  :description "Describe new-module here"
  :author "Anne N. O'Nymous"
  :license "GPLv3"
  :depends-on (#:stumpwm)
  :components ((:file "package")
               (:file "new-module"))) ; any other files you make go here
```

The `package.lisp` will contain:

```
(defpackage #:new-module
  (:use #:cl :stumpwm))
```

With these two files defined, and the implementation written in `new-module.lisp`, you should be able to load your module.

Before we load it, we have to add the path to our **load-path**. This can be accomplished by running the following from a REPL:

```
(stumpwm:add-to-load-path "~/.stumpwm.d/modules/new-module")
```

You can also run this interactively with `C-t ;`, which is bound to the `colon` command.

Because we've put our module in a sub-directory of the default **module-dir**, it will automatically get added to the **load-path** the next time StumpWM starts. If you choose to develop your module somewhere else (e.g. `~/quicklisp/local-projects`), then you'll have add

```
(add-to-load-path "~/quicklisp/local-projects/new-module")
```

to your `.stumpwmrc`.

When you've finished writing your module, you can distribute it however you see fit. If it becomes very popular, or you would like the StumpWM devs to maintain it (and they agree), you can have your module merged with the stumpwm-contrib repository on github, just open a pull request to start the discussion.

18 Hacking

18.1 Hacking: General Advice

1. Pay attention to file names and contents. If you're making changes to mode-line related code, don't put it in `core.lisp`. If you're introducing some completely new featureset, consider putting all of the new code in a new file.
2. Does a command need to be user-visible ("interactive") or is it just called by other commands?
   - If it's not going to be user-visible, you can just use the familiar `(defun foo () ...)` syntax.
   - If you want the command to be used interactively, you use StumpWM's `defcommand` syntax, as in the examples below.

     ```
     (defcommand test (foo bar)
        ((:string "How you're going to prompt for variable foo: ")
         (:number "How you want to prompt for variable bar: "))
        "This command is a test"
        (body...))

     (defcommand test2 () ()
        "This is also a test"
        (body...))

     (defcommand title (args) (interactive-args)
        "Doc string"
        (body...))
     ```

     So basically, inside the first set of parentheses after the function name, you specify what (if any) arguments will be passed to the command. The second set of parentheses tells StumpWM how to get those arguments if they're not explicitly passed to the command. For example,

     ```
     ((:string "What do you want to do: "))
     ```

     will read a string from the input the user provides. The quoted text is the prompt the user will see. Of course, if you were to, say, call the command test, as defined above, from another piece of code, it wouldn't give the prompt as long as you fed it arguments.
3. Note that all commands defined using the `defcommand` syntax are available both to be called with `C-t ;` and from within other lisp programs, as though they had been defun-ned (which, in fact, they have).
4. Any code that depends on external libraries or programs that some users might not have installed should be packaged as a module and placed in the `*data-dir*/modules/` directory.
5. Don't be afraid to submit your patches to the StumpWM mailing list! It may not immediately make it into the official git repository, but individual users might find it useful and apply it to their own setup, or might be willing to offer suggestions on how to improve the code.

18.2 Hacking: Adding Documentation and Editing This Manual

The manual is written in `texinfo`, so you may want to read that manual. The `stumpwm.texi.in` is processed by StumpWM with some additional markup in the form of three letter character entries at the beginning of a line. `@@@ function` defines functions, `%%% some-macro` expands to that macro and its docstring, etc. Contributors are strongly encouraged to add these items to this manual whenever something new is defined in a patch. You can test if your texinfo edits are valid by generating them with `make stumpwm.info`, and viewing the new `stumpwm.info` with `info -f /path/to/stumpwm.info`, or `make stumpwm.texi` for the raw stuff.

%%% macro
@@@ function
### variable
$$$ hook

!!! StumpWM command

18.3 Hacking: Using git with StumpWM

For quite a while now, StumpWM has been using the git version control system for development. If you're using one of the official releases, you can get the bleeding-edge source code from the official git repository with a single command:

```
$ git clone git@github.com:stumpwm/stumpwm.git
```

After this, you'll have a complete git repository, along with the complete revision history since the switch. Feel free to play around; git has some important features that actually make this safe!

Before we get to that stuff, though, you're going to want to tell git about yourself so that your information is included in your commits and patches. The very minimum you're going to want to do is:

```
$ git config --global user.name "Anne N. O'Nymous"
$ git config --global user.email "anonymous@foo.org"
```

Be sure to check out the manual for `git-config`–there are several options you might want to set, such as enabling colorized output or changing the editor and pager you use when making commits and viewing logs.

For the sake of argument, let's say you want to make some major changes to both `user.lisp` and `core.lisp`, add a file called `DANGEROUS_EXPERIMENT_DO_NOT_USE_OR_ELSE.lisp`, and remove the manual because you're too 1337 for such things. However, you don't want to break your entire StumpWM setup and start over. Thankfully, you don't have to. Before you get started, issue this command from the StumpWM source directory:

```
$ git checkout -b experimental
```

You should now find yourself in a new branch, called experimental. To confirm this, type `git branch`; there should be an asterisk next to the branch you're currently viewing. At any time, you can type `git checkout master` to return to your master branch, and at any time you can have as many branches of the project as you like. If you want to create a new branch based not on the master branch but on your experimental branch, for example, you'd type:

```
$ git checkout -b new-experiment experimental
```

This will place you in a newly-created branch called “new-experiment” which should be identical to your experimental branch as of the last commit (more on that soon). If you’re actually typing out the directions, switch back to your old experimental branch like so:

```
$ git checkout experimental
```

Anyway, now that you have a new branch, create that new file with the long name, which we’ll just call `danger.lisp` for brevity. Make whatever changes you want to it, and when you’re done, tell git about your new file.

```
$ git add dangerous.lisp
```

Now, let’s pretend you’re done making changes. Tell git you’re done for now:

```
$ git commit
```

This will open up a prompt in your editor of choice for you to describe your changes. Try to keep the first line short, and then add more explanation underneath (for an example, run the command `git log` and take a look at some of the longer commit explanations). Save that file and then do this:

```
$ git checkout master
$ ls
```

Then look for your new file. It’s not there! That’s because you’ve done all of your work in another branch, which git is currently hiding from you so that you can “check out” the branch called “master.” All is as it should be—your master repository is still safe.

```
$ git checkout experimental
```

Now, delete `manual.lisp` and `stumpwm.texi`. That’s right. Wipe them off the face of the Earth, or at least off the hard drive of your computer. When you’re done, you don’t have to tell git you’ve deleted them; it’ll figure it out on its own (though things may not compile properly unless you edit `Makefile.in` and `stumpwm.asd`. Anyway, go ahead and edit `core.lisp` and `user.lisp`. Really break ’em. Run free! When you’re done, do another commit, as above, and give it a stupid title like “lolz i b0rked stUmpwm guys wTF!?!?!!111!” Now try to compile. Just try. It won’t work. If it does, you’re some kind of savant or something. Keep up the good work. If you’ve actually managed to break StumpWM like you were supposed to, never fear! You have two options at this point.

One is to go back to the master branch (with another git checkout) and just delete your experimental branch, like so:

```
$ git branch -D
```

The “`-D`” means to force a delete, even if the changes you’ve made aren’t available elsewhere. A “`-d`” means to delete the branch if and only if you’ve merged the changes in elsewhere.

The other option is to create patches for each of your commits so far, delete the branch, and then apply any working/wanted patches in a new branch. Create your patches (after committing) like so:

```
$ git format-patch -o patches origin
```

(Before doing that you can review your changes with `git log origin..`)

You can also use the `format-patch` command to create a patch of working code to send in to the mailing list.

A developer might ask you to try out something they're working on. To fetch their master branch, you'd do this:

```
$ git remote add -f -m master -t master foo git://bar.org/~foo/stumpwm
```

Here, "foo" is the shorthand name you'll use to refer to that repository in the future. To checkout a local copy of that repository, you'd then do

```
$ git checkout --track -b foo-master foo/master
```

Later you could use `git pull foo` to update while looking at that branch (and note that `git pull` with no arguments, in the master branch, will update your StumpWM from the official repository).

Finally, if you want to move your experimental changes into your master branch, you'd checkout your master branch and run:

```
$ git merge experimental
```

If there are file conflicts, `git diff` will show you where they are; you have to fix them by hand. When you're done, do another

```
$ git commit -a
```

to finalize the changes to your master branch. You can then delete your experimental branch. Alternately, you can wait until your changes (assuming you sent them in) make it into the official repository before deleting your experimental branch.

18.4 Sending Patches

While patches are still welcome on the mailing list, StumpWM's development has mostly migrated to github's issue tracker. This means you can open a pull request to submit a patch to StumpWM. The following guidelines apply to pull requests and patches sent to the mailing list.

- Make sure it applies clean to the main git repository
- Ensure that you aren't introducing tabs, extra blank lines, or whitespace at the end of lines.
- Ensure your patch doesn't contain irrelevant indenting or reformatting changes.
- Try to make your patch address a single issue. If your patch changes two unrelated issues, break them into two seperate patches that can stand on their own.
- Don't send intermediate patches. When you're working on a feature you might make several commits to your local repository as you refine it and work out the bugs. When it's polished and ready to ship, send it as one patch! Sometimes it makes sense to send it as multiple patches if each patch contains a discrete feature or bug fix that can stand on its own. If one of your patches changes code that was added or modified in an earlier patch, consider merging them together and sending them as one.

19 Advanced Configuration

`*default-package*` [Variable]

This is the package eval reads and executes in. You might want to set this to `:stumpwm` if you find yourself using a lot of internal stumpwm symbols. Setting this variable anywhere but in your rc file will have no effect.

`*default-bg-color*` [Variable]

Default color for the desktop background.

`run-commands` **&rest** *commands* [Function]

Run each stumpwm command in sequence. This could be used if you're used to ratpoison's rc file and you just want to run commands or don't know lisp very well. One might put the following in one's rc file:

```
(stumpwm:run-commands
  "escape C-z"
  "exec firefox"
  "split")
```

`*startup-message*` [Variable]

This is the message StumpWM displays when it starts. Set it to NIL to suppress.

`*list-hidden-groups*` [Variable]

Controls whether hidden groups are displayed by 'groups' and 'vgroups' commands

`defprogram-shortcut` *name* **&key** (*command* **(string-downcase (string name))**) (*props* **(quasiquote (quote (class #S(comma :expr (string-capitalize command) :kind 0)))))** (*map* **(quote *top-map*)**) (*key* **(quasiquote (kbd #S(comma :expr (concat H- (subseq command 0 1)) :kind 0))))** (*pullp* **nil**) (*pull-name* **(intern1 (concat (string name) -PULL))**) (*pull-key* **(quasiquote (kbd #S(comma :expr (concat H-M- (subseq command 0 1)) :kind 0))))**) [Macro]

Define a command and key binding to run or raise a program. If *pullp* is set, also define a command and key binding to run or pull the program.

`*initializing*` [Variable]

True when starting stumpwm. Use this variable in your rc file to run code that should only be executed once, when stumpwm starts up and loads the rc file.

`loadrc` [Command]

Reload the `~/.stumpwmrc` file.

`*ignore-wm-inc-hints*` [Variable]

Set this to T if you never want windows to resize based on incremental WM_HINTs, like xterm and emacs.

`*max-last-message-size*` [Variable]

how many previous messages to keep.

`*module-dir*` [Variable]
: The location of the contrib modules on your system.

`set-module-dir` *dir* [Function]
: Sets the location of the for StumpWM to find modules

`*mouse-focus-policy*` [Variable]
: The mouse focus policy decides how the mouse affects input focus. Possible values are :ignore, :sloppy, and :click. :ignore means stumpwm ignores the mouse. :sloppy means input focus follows the mouse; the window that the mouse is in gets the focus. :click means input focus is transfered to the window you click on.

  If *MOUSE-FOCUS-POLICY* holds any value other than those listed above, mouse focus will behave as though it contains :IGNORE

`*resize-hides-windows*` [Variable]
: Set to T to hide windows during interactive resize

`*root-click-focuses-frame*` [Variable]
: Set to NIL if you don't want clicking the root window to focus the frame containing the pointer.

`*suppress-frame-indicator*` [Variable]
: Set this to T if you never want to see the frame indicator.

`*suppress-window-placement-indicator*` [Variable]
: Set to T if you never want to see messages that windows were placed according to rules.

`*text-color*` [Variable]
: The color of message text.

`*draw-in-color*` [Variable]
: When NIL color formatters are ignored.

`*timeout-frame-indicator-wait*` [Variable]
: The amount of time a frame indicator timeout takes.

`*top-level-error-action*` [Variable]
: If an error is encountered at the top level, in STUMPWM-INTERNAL-LOOP, then this variable decides what action shall be taken. By default it will print a message to the screen and to *standard-output*.

  Valid values are :message, :break, :abort. :break will break to the debugger. This can be problematic because if the user hit's a mapped key the ENTIRE keyboard will be frozen and you will have to login remotely to regain control. :abort quits stumpwm.

Command and Function Index

Variable Index