Emacs Init File

Keith Waclena

April 24, 2023

Contents

Introduction

This is my Emacs init file. It assumes Emacs version 28.2.

 This document is also available on the web. The tangled init file itself is also available.

This init file is maintained as an Org Mode document; the actual init file Elisp code is *tangled* from the Org file, and the HTML and PDF versions are generated from the same file.

```
;; init file proper begins here
```

Customization

Emacs has a very powerful (vaguely GUI) way to do customizations without writing Lisp code. By default, Emacs customizations go at the end of your init file; this is awkward for several reasons (e.g., because it complicates version control). So we arrange to use a separate file, `customizations.el`.

```
;; customizations
(setq custom-file (concat user-emacs-directory "customizations.el"))
(when (file-exists-p custom-file) (load custom-file 't))
```

Package Manager

The Emacs package manager let's you easily install additional packages from several repositories. Emacs autoloads all installed packages in subsequent Emacs sessions.

By default, the `package-archives` variable only contains the offical GNU ELPA repo and the new non-GNU ELPA repo, which between them contain about 500 additional packages. The main third-party repo is Melpa; it is divided into a stable archive and an unstable one; I much prefer melpa-stable. Unfortunately, some packages in melpa-stable depend on packages that are only in the unstable repo, so I need to use both. I set `package-archive-priorities` to make sure I get stable versions when possible

I also add my own personal archive here (kw) (packaging Emacs Lisp code is very easy).

```
;; package manager
(with-eval-after-load 'package
  (dolist (arc '(("melpa-stable" . "http://stable.melpa.org/packages/")
 ("melpa" . "https://melpa.org/packages/")
 ("kw" . "https://www2.lib.uchicago.edu/keith/software/emacs/packages/")))
    (add-to-list 'package-archives arc t))
  (setq package-archive-priorities '(("kw" . 11) ("gnu" . 10) ("nongnu" . 9) ("melpa-stable" . 8)))
  (add-hook 'package-menu-mode-hook 'hl-line-mode))
```

use-package is John Wiegley's amazing macro for lazily loading and configuring packages. I use it heavily, but it doesn't come with Emacs! So I also define a fallback noop in case it's unavailable, to prevent my init file from blowing up.

Frankly, I'm going to move away from use-package because of the hassle involved in using it when it doesn't come with Emacs.

```
(unless (fboundp 'use-package)
  (message "`use-package' not available!")
  (defmacro use-package (name &rest args)
    (message "%S disabled because `use-package' unavailable." name)))
```

Appearance

Colors and Fonts

I prefer a dark theme (white-on-black). Emacs has many fancy color themes, but I haven't taken the time to evaluate all of them… I tweak the exact values based on hostname because the brightness of my various monitors varies.

I also specify my preferred monospaced font, Terminus.

```
(condition-case err
    (cond
     ((string= "jfcl" (system-name))
      (set-face-attribute 'default nil
  :font "-Xos4-Terminus-normal-normal-normal-*-16-*-*-*-m-*-iso10646-1"
  :foreground "white" :background "gray12"))
     ((string= "beisl" (system-name))
      (set-face-attribute 'default nil
  :font "-Xos4-Terminus-normal-normal-normal-*-20-*-*-*-m-*-iso10646-1"
  :foreground "white" :background "gray8"))
     (t
      (set-face-attribute 'default nil :foreground "white" :background "black")))
  (error
   (message "%s" err)
   (set-face-attribute 'default nil :foreground "white" :background "black")))
```

I use `hl-line-mode` heavily and like the highlighted line to be just a lighter version of my frame's background color. (I should probably try to compute this color automagically, but for now it has to be kept in sync with the above.)

```
(defface kw-hl-line
  '((t :inherit hl-line :extend t :background "gray36"))
  "My face for highlighting the current line in Hl-Line mode."
  :group 'minimacs)
(setq hl-line-face 'kw-hl-line)
```

I bind the `text-scale-adjust` to the otherwise undefined `C-+`; my first step in tweaking font size is always to increase it...

```
(global-set-key [(control +)] 'text-scale-adjust)
```

Minimize GUI Elements

I avoid using the mouse as much as possible, so I don't have any use for the menu bar and the tool bar — for me they just waste real estate. The same goes for the scroll bars.

```
(when (fboundp 'menu-bar-mode) (menu-bar-mode -1)) ;turn off menu bar
(when (fboundp 'tool-bar-mode) (tool-bar-mode -1)) ;turn off tool bar
;;turn off vertical scroll bar
(when (fboundp 'scroll-bar-mode) (scroll-bar-mode -1))
;;turn off horizontal scroll bar
(when (fboundp 'horizontal-scroll-bar-mode) (horizontal-scroll-bar-mode -1))
;; turn off GUI dialogs...
(setq use-file-dialog nil use-dialog-box nil)
(blink-cursor-mode -1)  ; blinking cursor drives me nuts
;; tooltips are useless & evil over slow network connections
(tooltip-mode -1)
(setq visible-bell t)   ; audible bell too noisy
```

Buffer Boundaries and Empty Lines

I use a tiling window manager, which can result in windows being very skinny; as my default, I prefer truncated lines to wrapped lines.

```
(setq-default truncate-lines t) ; good for tiling window managers
```

Indicate in the right fringe if we're seeing the top or bottom of the buffer; this plus `sml-modeline` (see Packages, below) completely replace the utility of a vertical scrollbar (since I never use it to scroll via the mouse, it's only utility is as a graphical indication of the buffer position).

```
;; indicate buffer boundaries in the right fringe
(setq-default indicate-buffer-boundaries
      '((top . right)
(bottom . right)
(t . nil)))
```

When the frame has more display lines than the buffer has lines of text, I want to see graphically that these aren't really blank lines of text at the end of my file, but I only need this in major modes that are used for editable files.

```
(defun kw-turn-indicate-empty-lines-on ()
  "Set 'indicate-empty-lines' to t."
  (setq indicate-empty-lines t))
;; indicate empty lines at end of buffer in modes for editable files
(add-hook 'prog-mode-hook   #'kw-turn-indicate-empty-lines-on)
(add-hook 'text-mode-hook   #'kw-turn-indicate-empty-lines-on)
```

Modeline

These tweaks add more information to the modeline, in particular the size of the file and the time of day.

```
(size-indication-mode 1)            ; how big is that file?
(display-time)                      ; what time is it?
(display-battery-mode)              ; how long have I got?
```

Biffing (Email Notification) in the Mode Line This is probably extremely rare these days, but I have my email delivered to my desktop, and this code puts a notification in the modeline whenever new mail comes in (we used to call this "biff"). I tweak the color to make it more noticeable.

```
(setq display-time-use-mail-icon t      ; for graphical display
      display-time-mail-string "")    ; for text display: unicode #x1f4e7 email symbol
(defface display-time-mail-face '((t (:background "red")))
  "If display-time-use-mail-icon is non-nil, its background
colour is that of this face.  Should be distinct from
mode-line. Note that this does not seem to affect
display-time-mail-string as claimed."
  :group 'minimacs)
(setq display-time-mail-face 'display-time-mail-face) ;surely this is bogus?
```

Inhibit the Startup Screen

It's nice, but I've seen it.

```
(setq inhibit-startup-screen t)
```

Basics

My identity:

```
(when (string= "" user-full-name) (setq user-full-name "Keith Waclena"))
```

`ffap` (`find-file-at-point`) can be too optimistic about remote file names.

```
;; ffap can try to ping hostname-like things which can take forever to timeout...
(mapc
 (lambda (var) (set var 'reject))
 '(ffap-machine-p-local ffap-machine-p-known ffap-machine-p-unknown))
```

Emacs does authentication for various things (e.g. mail) via files named in `auth-sources`.

```
(setq auth-sources '(,(expand-file-name "~/emacs/authinfo.gpg"))) ; authentication
(add-to-list 'auto-mode-alist '(,(rx bos (eval (expand-file-name (car auth-sources))) eos) . authinfo-mode
```

I have piles of old Emacs Lisp code in my `home-emacs` directory, so I add that to `load-path`. Really, I'm trying to eliminate as much of this old code as possible.

```
(defconst home-emacs (file-name-as-directory (expand-file-name "~/emacs"))
  "My Emacs Lisp code lives here.")
(add-to-list 'load-path home-emacs)
```

I make cutting-and-pasting use both the X primary selection and the clipboard. I have to admit I'm not totally happy with this approach.

```
(setq save-interprogram-paste-before-kill t yank-pop-change-selection t)
(setq select-enable-primary t select-enable-clipboard t)
```

I run Emacs as a daemon via `(server-start)`, so I like having my home directory as working directory in the `*scratch*` buffer, and, most importantly, Gnus buffers (which get started from the `*scratch*` buffer).

```
(cd "~")
```

I *never* want TAB to insert a TAB character.

```
(setq-default tab-always-indent t)
```

Now we define a handy keymap on `F12` for personal commands.

```
(defvar kw-handy-keymap (make-sparse-keymap)
  "*Handy f12 keymap.")
(global-set-key (kbd "<f12>") kw-handy-keymap)
```

Very General Command Aliases

```
;; very general M-x aliases
(defalias 'qrr 'query-replace-regexp)
(defalias 'ffr 'find-file-read-only)
(defalias 'firefox 'browse-url-firefox)
```

Operating System Customizations

I like to use the same init file whether I'm running under Linux, FreeBSD, Mac OS, ChromeOS, or even Windows!. Some OS-specific tweaks are necessary.

```
(cond
 ((eq 'gnu/linux system-type)
  ;; I run Linux on a Macbook and want the command key for Meta
  (setq x-super-keysym 'meta))
 ((eq 'darwin system-type)     ; homebrew
  (add-to-list 'exec-path "/usr/local/bin")))
```

Extended Commands

`C-x ESC ESC` (`repeat-complex-command`) needs a bigger history and also a more intuitive activation keystroke.

```
(with-eval-after-load 'chistory
  (setq list-command-history-max 120)
  (define-key command-history-mode-map (kbd "<return>") #'command-history-repeat))
```

Paragraph Scrolling

I really like to scroll windows by paragraphs; it helps prevent me from losing my place. I bind these functions to various keystrokes in many modes.

```
(defun kw-scroll-one-paragraph (&optional arg)
  "Scroll forward by ARG paragraphs."
  (interactive "P")
  (let ((here (point)))
    (move-to-window-line 0)
    (forward-paragraph arg)
    (re-search-forward "\\=[[:space:]\n]+" nil t)
    (forward-line -1)
    (recenter 0)
    (when (> here (point))
```

```
      (goto-char here))))

(defun kw-scroll-one-paragraph-back (&optional arg)
  "Scroll backward by ARG paragraphs."
  (interactive "P")
  (kw-scroll-one-paragraph (or arg -1)))
```

Help

I like to scroll by paragraphs in Help Mode.

```
(with-eval-after-load 'help-mode
  (define-key help-mode-map "[" #'kw-scroll-one-paragraph-back)
  (define-key help-mode-map "]" #'kw-scroll-one-paragraph))
```

Calendar, Clock, and Diary Settings

The Emacs calendar and diary are very useful; many other packages (maps, weather report, . . .) will use some of these settings as well.

I use a combination of Emacs diary files and the Org Mode agenda; I want notifications in Emacs via the Echo Area, and also via my Unix desktop.

```
(setq world-clock-list
      '(("America/Chicago" "Chicago")
("America/New_York" "Kalamazoo")
("America/New_York" "Syracuse")
("Europe/Vienna" "Vienna")))
(setq calendar-location-name "Chicago, IL" ;your default location
      calendar-latitude 41.795088232902785
      calendar-longitude -87.5813341140747
      calendar-mark-holidays-flag t        ;colorize holidays in the calendar
      calendar-mark-diary-entries-flag t   ;also diary entries
      appt-display-format 'echo            ;appointment notifications in echo area
      holiday-bahai-holidays nil           ;these religions have MANY holidays
      holiday-islamic-holidays nil         ;... that I don't get off
      diary-file (expand-file-name "~/notes/diary")
      diary-comment-start "(*"             ;comments allowed in diary
      diary-comment-end   "*)")
(add-hook 'diary-list-entries-hook         ; recursive diary files are good
  'diary-include-other-diary-files)
(add-hook 'diary-mark-entries-hook         ; ... so mark them too
  'diary-mark-included-diary-files)
(appt-activate 1)                          ; appointment notifications, please
(require 'notifications)              ;also via desktop notifications
```

```
;; notable holidays that Emacs doesn't know about
(with-eval-after-load 'holidays
  (add-to-list 'calendar-holidays
       '(holiday-sexp
 '(if (zerop (% year 4))
      (calendar-gregorian-from-absolute
       (1+ (calendar-dayname-on-or-before
    1 (+ 6 (calendar-absolute-from-gregorian
    (list 11 1 year)))))))
 "US Presidential Election")))
```

Performance and Behavior

Increasing the `gc-cons-threshold` makes things a little more quiet.

```
(setq gc-cons-threshold             20000000) ; 20MB means less gc
```

I don't want to delete the active region with a mere `DEL`; having to type `C-w` as god intended is fine. Additionally, the default setting for `select-active-regions` (`t`) is way too eager, automatically updating the X selection. Since one frequently sets the active region around the entirity of an enormous buffer, this can cause a noticeable delay, especially if running X over a slow network connection. I prefer to explicitly type `M-w`.

```
(setq delete-active-region nil)         ; too GUIish
(setq select-active-regions 'only)      ; too eager
```

"Many Emacs modes struggle with buffers which contain excessively long lines, and may consequently cause unacceptable performance issues."

```
(when (version<= "27.1" emacs-version)    ;speed up long lines
  (global-so-long-mode 1))
```

Emacs's impressive bidirectional text handling necessitates a lot of checks on each line, and these checks slow things down. As a stereotypical monoglot English speaker, I have the luxury to just turn these off.

```
(setq-default bidi-paragraph-direction 'left-to-right)
(when (version<= "27.1" emacs-version)
  (setq bidi-inhibit-bpa t))
```

Enabling and Disabling

Emacs disables certain commands "to prevent users from executing [them] by accident; we do this for commands that might be confusing

to the uninitiated". If you invoke a disabled command, Emacs will ask you if it should be enabled permanently, and if so, will save commands like these to your init file. Here are all the ones I've enabled over the years.

Relatedly, recursive minibuffers (more a concept than a command) are really useful, but are disabled by default.

```
;; I like to use these commands...
(dolist (cmd '(scroll-left downcase-region upcase-region
       narrow-to-region narrow-to-page eval-expression
       set-goal-column))
  (put cmd 'disabled nil))
;; ... and also recursive minibuffers
(setq enable-recursive-minibuffers t)
```

In addition, I explicitly disable certain commands that Emacs doesn't! These are commands that I frequently invoke by accident because I can't tell them apart from the similar commands I really want to invoke!

```
;; i want to always use -pgpmime flavors, but can't remember its keybindings...
(with-eval-after-load 'mml-sec
  (dolist (cmd '(mml-secure-encrypt-pgp
 mml-secure-encrypt-smime
 mml-secure-message-encrypt-pgp mml-secure-message-encrypt-smime
 mml-secure-sign-pgp mml-secure-sign-smime mml-secure-message-sign-pgp
 mml-secure-message-sign-smime))
    (put cmd 'disabled t)))
```

I never use these mouse bindings intentionally, but I constantly invoke them by accident (stupid touchpad!).

```
;; thanks to my touchpad I'm always accidentally invoking these commands
;; e.g. pasting with mouse-wheel click, text scale adjust...
(unless (string= "jfcl" (system-name))
  (condition-case nil
      (progn
(global-disable-mouse-mode)
;; I know it's enabled...
(setq global-disable-mouse-mode-lighter nil))
    (t
     (dolist (k '("<C-down-mouse-5>" "<C-mouse-5>" "<C-double-down-mouse-5>"
  "<C-double-mouse-5>" "<C-triple-down-mouse-5>" "<C-triple-mouse-5>"
  "<C-down-mouse-3>" "<C-wheel-down>" "<mouse-2>"))
       (define-key (current-global-map) (kbd k) nil)))))
```

```
;; TODO move to dot.gnus!
(put 'gnus-draft-send-all-messages 'disabled t) ;a nightmare scenario!
(put 'gnus-topic-indent 'disabled t)            ;I hate it when this happens
```

Finally, in graphical-mode emacs (and with a tiling WM) suspending makes no sense.

```
(put 'suspend-frame 'disabled t)
```

Client / Server Emacs

I run Emacs in `server-mode`: that is, I start up a single Emacs process when I login to my computer, and keep it running until I need to reboot for an OS kernel update (typically weeks for a rolling-release OS like Arch Linux; much longer for other systems). I connect to this Emacs via `emacsclient` from various terminals on my computer, and I also connect to the Emacs running on my work desktop over the network via an ssh'd `emacsclient` from my home laptop.

`(server-start)` will error if there's already a server running on this machine, and I don't want two servers for any reason (extremely confusing). But sometimes I need to start up a fresh Emacs (not a client) to test my init file, so we need to ignore the error.

```
(require 'server)
(if (fboundp 'server-running-p)         ; new in v28?
    (unless (server-running-p)
      (server-start))
  ;; server-start will fail if the server is already running...
  (ignore-errors (server-start)))
```

It's almost always a mistake to try to terminate the Emacs server, so I set up exit confirmation, and also save and restore the state of my long-running Emacs whenever I do need to kill it.

```
(setq confirm-kill-emacs 'yes-or-no-p)        ; exiting emacs is silly
;; restore state on the rare occasion that I have exited...
(when (string= "jfcl" (system-name))
  (setq desktop-restore-frames nil))       ; I don't want frames restored on jfcl
(setq desktop-restore-eager 12)         ; be lazier for quicker startup
(desktop-save-mode 1) ; startup emacs with previous state (files, buffers)
(save-place-mode 1)   ; come back to where we were in that file
```

Conversely, a long-running Emacs tends to collect ancient buffers that one hasn't been using for weeks; these take up memory, but worse, embiggen the completion space. Midnight Mode deletes such buffers at midnight (only if they've been saved, of course).

```
(require 'midnight)
(midnight-mode 1)     ; clean up old buffers at midnight
;; some buffers should never be cleaned up
(dolist (buf '("*Calculator*" "*Calc Trail*" "*quick*" ".newsrc-dribble"))
  (add-to-list 'clean-buffer-list-kill-never-buffer-names buf))
(dolist (rex `(,(rx ".epub" eos) ,(rx bos "bbdb" eos)))
  (add-to-list 'clean-buffer-list-kill-never-regexps rex))
;; proced continuously updates its buffer htop-style in the backgground (I think?) and thus increases the
(add-to-list 'clean-buffer-list-kill-buffer-names "*Proced*")
;; CANDIDATES FOR DELETION
;; ^ \*diff-syn
;; *sent wide reply to
;; *sent mail to
;; *Org Help*
;; ^ *RNC Input<9>*
;; diff-mode leaves a million of these and they can be HUGE
(add-to-list 'clean-buffer-list-kill-regexps (rx bos " *diff-syntax:"))
;; TODO move to dot.gnus
(dolist (rex `(,(rx bos "*Group*" eos) ,(rx bos "*Summary ")))
  (add-to-list 'clean-buffer-list-kill-never-regexps rex))
```

Window Management

Emacs pops up many windows with few lines in them (help windows and the like). By default these windows take up half your screen. I like these windows to have their size minimized. `temp-buffer-resize-mode` automates this (though it has some quirks).

```
;; windows and scrolling
(setq window-min-height 2)            ; small windows save real estate
(temp-buffer-resize-mode 1)           ; the greatest thing ever! or is it!?
```

`winner-mode` makes it easy to restore the window configuration after transient windows (like `*Help*`) have changed it.

```
(winner-mode 1)                       ; C-c <left> restores previous window config
;; add more felicitous bindings
(define-key winner-mode-map [(control c) (control left)] 'winner-undo)
(define-key winner-mode-map [(control c) (control right)] 'winner-redo)
```

I use a tiling window manager, so frequently my Emacs is temporarily very skinny; by default, lines longer than the window width will wrap, which I find confusing, so I prefer to (visually) truncate the lines. (Except in certain buffers that display the values of Emacs variables, which are often very long lists.)

```
(setq-default truncate-lines t)       ; good for tiling window managers
```

```
; but not in Elispy buffers that display long sexprs
(dolist (hook '(help-mode-hook ielm-mode-hook))
  (add-hook hook (lambda () (setq truncate-lines nil))))
```

Since I don't use the mouse, `windmove` gives me nice directional versions of `C-x o` for switching windows, bound to `S-left`, `S-up`, etc.

```
(windmove-default-keybindings)        ; S-{left,right,up,down} switches windows
```

I set up several key-bindings to shrink and enlarge windows, horizontally and vertically.

```
(global-set-key [(control {)]   'shrink-window-horizontally)
(global-set-key [(control })]   'enlarge-window-horizontally)
(global-set-key "\C-x\C-^"      'enlarge-window)
(global-set-key "\C-^"          'enlarge-window)
(defun kw-shrink-other-window ()
  "Minimize the other window."
  (interactive)
  (let ((win (selected-window)))
    (save-excursion
      (other-window 1)
      (shrink-window-if-larger-than-buffer)
      (select-window win))))
(global-set-key "\C-x+"         'kw-shrink-other-window)
(global-set-key "\M-\C-^"       'shrink-window)
```

When splitting a window, the default is to split it in half. If you split the new half window, it's now ¼ the original size. Instead I prefer the space to be taken proportionately from all the windows in the frame, so three splits results in three equal-sized windows each the size of the frame.

```
(setq window-combination-resize t)      ;split windows proportionately
```

This Hydra makes it easy to rearrange the windows in a frame.

```
(use-package transpose-frame :defer t
  :commands
  (transpose-frame
   flip-frame
   flop-frame
   rotate-frame
   rotate-frame-clockwise
   rotate-frame-anticlockwise))

(with-demoted-errors "%s"
  (let ((key "C-x |"))
```

```
    (use-package hydra)
    (global-set-key
     (kbd key)
     (condition-case err
 (defhydra kw-transposing (:hint nil)
   "\n_t_, _|_:  transpose _v_:  vert flip _h_:  horiz flip _8_:  180° _9_:  90° _0_:  90° "
   ("t" transpose-frame)
   ("|" transpose-frame)
   ("v" flip-frame)
   ("h" flop-frame)
   ("8" rotate-frame)
   ("9" rotate-frame-clockwise)
   ("0" rotate-frame-anticlockwise)
   ("q" nil "exit"))
       (error
(message "warning: %s unbound: %S" key err)
nil)))))
```

Buffer Management

Window Management isn't complete without *buffer management*. 95% of the time,the *Buffer Stack* is a better way to switch buffers than the usual `C-x b` (`switch-to-buffer`). Certain buffers should be excluded from the buffer stack because I never want to switch to them. Finally, I have my own preferred key bindings.

Breaking News: the Buffer Stack seems to be obsolete! The package isn't in any of the usual places. While I have a personal copy that I could add to my own package repo, I'm going to experiment with replacing it with the newer built-ion commands `previous-buffer` and `next-buffer`.

```
;; simulating my old buffer-stack-untracked settings
(setq switch-to-prev-buffer-skip
      (lambda (_win buf _bury)
(member (buffer-name buf)
'("*Pinentry*" "*Ediff Registry*" "*log-edit-files*"
  "*Org Help*" "*Org PDF LaTeX Output*" "*Ibuffer*"
  ".newsrc-dribble" "*caml-help*" "*Pp Eval Output*"))))
;; my old buffer stack bindings
(dolist (pair '(("C-S-k"        . previous-buffer)
("C-c <down>"   . previous-buffer)
("C-c C-<down>" . previous-buffer)
("<C-down>"     . previous-buffer)
("C-c <up>"     . next-buffer)
```

```
("C-c C-<up>"   . next-buffer)
("<C-up>"       . next-buffer)))
  (global-set-key (kbd (car pair)) (cdr pair)))

;;; buffer-stack
(ignore
 '(use-package buffer-stack
    :config
    (setq buffer-stack-show-position 'buffer-stack-show-position-buffers)
    (setq buffer-stack-untracked (remove "*Group*" buffer-stack-untracked))
    ;; I never want to switch to these buffers
    (dolist (b '("*Pinentry*" "*Ediff Registry*" "*log-edit-files*"
 "*Org Help*" "*Org PDF LaTeX Output*"
 ".newsrc-dribble" "*caml-help*" "*Pp Eval Output*"))
      (add-to-list 'buffer-stack-untracked b))
    :bind
    (("C-S-k" . buffer-stack-down)
     ("C-c <down>" . buffer-stack-down)
     ("C-c C-<down>" . buffer-stack-down)
     ("<C-down>"   . buffer-stack-down)
     ("C-c <up>"   . buffer-stack-up)
     ("C-c C-<up>"   . buffer-stack-up)
     ("<C-up>"     . buffer-stack-up))))
```

I have a love-hate relationship with *ibuffer*: the concept is great — a Dired for buffers, rather than files — but I think I need to sit down and completely redo all the key bindings…

```
;; ibuffer: dired for buffers
(global-set-key (kbd "C-x C-b") 'ibuffer)
(with-eval-after-load 'ibuffer
  (define-key ibuffer-mode-map (kbd "/ d") 'ibuffer-filter-by-directory))
(defun kw-ibuffer-recent-buffer (old-ibuffer &rest arguments)
  "Open ibuffer with cursor pointed to most recent buffer name."
  (let ((recent-buffer-name (buffer-name)))
    (apply old-ibuffer arguments)
    (with-demoted-errors "FYI: %s"
      (ibuffer-jump-to-buffer recent-buffer-name))))
(advice-add #'ibuffer :around #'kw-ibuffer-recent-buffer)
(setq ibuffer-default-shrink-to-minimum-size t)
```

The new `C-x x` keymap of buffer commands can hold some additional commands.

```
(global-set-key (kbd "C-x x k") #'kill-buffer)
```

File Management

In my opinion, this is the correct setting for Unix users:

```
(setq backup-by-copying-when-linked t)        ; make hard links stick
```

Emacs automatically saves your work periodically as you edit; I like it to be more aggressive to minimize the amount of work lost in a crash. To reduce the frequency of a helpful warning, I increase the `large-file-warning-threshold` by an order of magnitude.

```
;; computers are fast now
(setq auto-save-interval 20 ; default: 300
      auto-save-timeout 30) ; default: 960
(setq auto-save-no-message t) ; I trust you
;; computers are big now
(setq large-file-warning-threshold 100000000) ; 100MB
```

I like `recentf-mode` which remembers files that you've opened recently, but it's normally only available from the Emacs menu, which I don't use. My `find-recent` command makes use of it.

```
(recentf-mode 1)                      ; keep track of recently opened files
(setq recentf-max-saved-items 100)
;; tramp file names cause completion to raise pointless errors if, say, the remote is down
(push (lambda (fn) (string-match tramp-file-name-regexp fn)) recentf-exclude)
```

View Mode

When opening a file that's read-only, turn on `view-mode` automatically! I like to move by paragraphs almost everywhere.

```
(setq view-read-only t)
(with-eval-after-load 'view
  (define-key view-mode-map "}" #'forward-paragraph)
  (define-key view-mode-map "{" #'backward-paragraph)
  (define-key view-mode-map "[" #'kw-scroll-one-paragraph-back)
  (define-key view-mode-map "]" #'kw-scroll-one-paragraph))
```

Scrolling

Since I suppress the horizontal scroll bar, I need a pair of single-keystroke commands to replace it.

```
;; horizontal scrolling
(global-set-key [(control <)]   'scroll-left)
(global-set-key [(control >)]   'scroll-right)
```

I do a lot of incremental scrolling, often one line at a time; the default scroll settings cause the window to jump more than I like. I need a pair of keystrokes for these scrolling functions; since I never suspend my Emacs, `C-z` is available (the downside: this means I'm *constantly* suspending other people's Emacsen when I sit down at their desk).

```
;; Single-line scrolling
(setq scroll-margin 0 scroll-conservatively 100000) ; make scrolling less jumpy

(defun scroll-down-line (arg)
  "Scroll text of selected window downward ARG lines; or one line if no ARG.
If ARG is omitted or nil, scroll downward by one line."
  (interactive "p")
  (scroll-up-line (if arg (- arg) -1)))

(global-set-key "\C-z"    #'scroll-up-line)
(global-set-key "\M-\C-z" #'scroll-down-line)
```

I also like to scroll by paragraphs, especially in text and email buffers and in the web browser.

```
(global-set-key [(control meta })] 'kw-scroll-one-paragraph)
(global-set-key [(control meta {)] 'kw-scroll-one-paragraph-back)
```

International Character Set Support

`C-\` toggles the current multilingual input method. As a monoglot English speaker, I use it to enter the occasional French, German, or Spanish word, so `latin-1-postfix` works nicely for me.

```
(setq default-input-method "latin-1-postfix") ; default input method for C-\
```

Words and Language

Spelling Correction

Spelling correction requires an external program; the two most common choices are `aspell` and `hunspell`. I can't get `hunspell` to handle contractions, so I prefer `aspell`, but `hunspell` is better than nothing, so I'm willing to use it as a fallback.

```
(setq ispell-program-name
      (or (executable-find "aspell") (executable-find "hunspell")))
(setq ispell-dictionary "american")
```

By default, Emacs will ask for confirmation every time you add a word to your personal dictionary. This is nuts.

```
(setq ispell-silently-savep t) ; always assume it's okay to save the dictionary
```

The best way to do spelling correction is with `flyspell`. It even has a special mode for programming language modes (where it only checks spelling in comments).

```
(add-hook 'text-mode-hook #'flyspell-mode) ;spell-check in text mode
(add-hook 'prog-mode-hook #'flyspell-prog-mode) ; spell-check comments and strings
```

However, `flyspell` is *far* too aggressive in its key bindings, stealing `M-TAB` (a.k.a `C-M-i`), which is way too important a binding in too many modes, especially when `flyspell` already binds the handy `C-.` for the same thing.

```
(with-eval-after-load 'flyspell
  (define-key flyspell-mode-map (kbd "C-c $") nil)
  (define-key flyspell-mode-map (kbd "<M-tab>") nil)
  (define-key flyspell-mode-map (kbd "C-M-i") nil))
```

TODO *Thesaurus*

```
(define-key kw-handy-keymap (kbd "t") 'mw-thesaurus-lookup-dwim)
(with-eval-after-load 'mw-thesaurus
  (add-hook 'mw-thesaurus-mode-hook 'visual-line-mode))
```

Tramp

Given that what Tramp does is amazing complex, it's astonishing how straightforward it is to use! Except for one thing. It needs to be able to recognize your shell prompt. And it's really good at recognizing shell prompts.

But not my stupid shell prompt...

I also turn off `tramp-completion-use-auth-sources` because it causes me to have to type my GPG passphrase, and I don't need completion for hostnames or usernames.

```
(setq shell-prompt-pattern "^[^#$%>;\n]*[#$%>;] .*") ; my shell prompt is weird
(setq tramp-completion-use-auth-sources nil)         ; very interruptive with selectrum
(setq tramp-allow-unsafe-temporary-files t)          ; they're not really unsafe...
```

Grep

The first line of a `grep-mode` buffer is typically a `grep` command that's so long it takes up a dozen visual lines, so I often can't see the first hit without scrolling... (This problem seems to be mooted by the recent addition of ellipsisization to `grep`.)

```
(add-hook 'grep-mode-hook (lambda () (setq truncate-lines t)))
```

I find it very useful to be able to flush and keep lines in the *grep* buffer—useful enough to have key bindings—but the buffer is read-only, so `flush-lines` and `keep-lines` don't work.

```
(defun kw-keep-or-flush (func)
  (save-excursion
    (let ((inhibit-read-only t))
      (goto-char (point-min))
      (save-restriction
(when (text-property-search-forward 'mouse-face)
  (narrow-to-region (point-at-bol) (point-max)))
(funcall func nil nil nil t)))))

(add-hook 'grep-mode-hook
  (lambda ()
    (define-key grep-mode-map (kbd "f")
      (lambda (regexp &optional rstart rend interactive)
(interactive (keep-lines-read-args "Flush lines containing match for regexp"))
(kw-keep-or-flush #'flush-lines)))))
(add-hook 'grep-mode-hook
  (lambda ()
    (define-key grep-mode-map (kbd "k")
      (lambda (regexp &optional rstart rend interactive)
(interactive (keep-lines-read-args "Keep lines containing match for regexp"))
(kw-keep-or-flush #'keep-lines)))))
```

Amazingly, the default `m` alias doesn't include `GNUmakefile`! Also, I use the file extension `.db` for Refer databases. So I use these file aliases for `lgrep` and `rgrep`.

```
(with-eval-after-load 'grep
  (add-to-list 'grep-files-aliases '("m" . "GNUmakefile [Mm]akefile*"))
  (add-to-list 'grep-files-aliases '("db" . "*.db")))
```

I prefer `egrep` to the Emacs default of `grep`; the `--color` option results in colorized hits.

```
(with-eval-after-load 'grep
  (grep-compute-defaults)
  (grep-apply-setting 'grep-command "grep --color -EinH --null -e ")
  (grep-apply-setting 'grep-template (replace-regexp-in-string "\\<grep\\>" "grep -E" grep-template))
  (grep-apply-setting 'grep-find-command "find . -type f -exec grep --color -EniH --null -e  \\{\\} +"))
```

Since I use Zsh as my `system-shell`, and by default Zsh throws an error if a glob pattern doesn't match any files, I need to replace

the default `..?*` glob pattern in the grep `all` file alias with one that won't blow up.

```
(with-eval-after-load 'grep
  ;; the "..?*" in grep-files-aliases blows up (in the typical dir with no match) with system-shell "zsh".
  ;; (N): zsh glob qualifier (zshexpn(1)) prevents .[!.]* from blowing up if there are no dot files
  (setf (alist-get "all" grep-files-aliases nil nil 'string-equal) "* .[!.]*(N)"))
```

Info and Man Pages

I set up `C-h C-i` to open an Info manual by name. (This is now available in v28 as `C-h R` but I'm really used to my old binding.)

```
;;; info bindings
(define-key 'help-command (kbd "C-i") 'info-display-manual)
```

I add my personal Info directory to Info's list of directories. I also dedupe the lists because it is initialized via several sources (by Emacs itself, by me, and by the Arch Linux package manager).

I also use Tun-Anh Nguyn's `info-colors` package to make Info pages more readable, and Bruno Félix Rezende Ribeiro's `info-rename-buffer` package to name Info buffers after their contents (absolutely essential).

```
(with-eval-after-load 'info
  (add-to-list 'Info-additional-directory-list (concat home-emacs "info"))
  (delete-dups Info-additional-directory-list) ; destructive
  (delete-dups Info-directory-list)
  (with-demoted-errors "%s"
    (info-rename-buffer-mode +1))
  (add-hook 'Info-selection-hook 'info-colors-fontify-node))
```

In `Man-mode` and `Info-mode`, I want my usual paragraph-scrolling commands:

```
(dolist (pair '((Man-mode-hook . Man-mode-map) (Info-mode-hook . Info-mode-map)))
  (add-hook (car pair)
    `(lambda ()
       (define-key ,(cdr pair) "[" #'kw-scroll-one-paragraph-back)
       (define-key ,(cdr pair) "]" #'kw-scroll-one-paragraph))))
```

In `Man-mode` I want `goto-address-mode`:

```
(add-hook 'Man-mode-hook #'goto-address-mode)
```

Incremental Search and Occur

It's useful to be able to scroll and adjust the visible part of the buffer while in the midst of a search. `char-fold-to-regexp` is essential in a

Unicode environment (so searching for "cafe" will find "café").

```
;; incremental search
(setq isearch-allow-scroll t)           ; scroll while searching
(setq search-default-mode 'char-fold-to-regexp) ; cafe = café
```

In `occur-mode`, I find that highlighting the entire line as I scroll is very useful.

```
(add-hook 'occur-mode-hook 'hl-line-mode)
```

Completion

Completion is key to the usablility of commands like `M-x`, `find-file`, `switch-to-buffer`, etc. The aggressive `initials` style lets you type `M-x buf` and have it expand to `M-x browse-url-firefox`, for example. You may want to play around with the order of these to get behavior that you like. Emacs says, "Note that `completion-category-overrides` may override these styles for specific categories, such as files, buffers, etc."

There are many fancier completion systems available via the package manager, such as the popular Helm (way too big and complex for me). I use Selectrum.

```
;; completion
(setq completion-styles '(partial-completion substring flex))
(setq read-file-name-completion-ignore-case t)
(setq read-buffer-completion-ignore-case t)
(with-demoted-errors "%S"
  (selectrum-mode +1)
  (selectrum-prescient-mode +1)
  (prescient-persist-mode +1)
  (marginalia-mode +1))
(setq selectrum-show-indices t)
;; go to completions window; nice if selectrum's not available
(define-key minibuffer-local-map [?\M-C] #'switch-to-completions)
;; pointless with good completion, and can erase useful messages
(setq extended-command-suggest-shorter nil)
```

Abbrevs

One of the oldest Emacs features is *Abbrevs*, a system by which you can define a given string as an abbreviation for some longer string. I've never found this to be useful, due to the effort of having to explicitly define each abbrev.

`dabbrev-expand` is a more recent (being only 22 years old, as opposed to 33) and much more usable alternative, which simply expands the previous string "dynamically" to the most recent preceding word for which it is a prefix; repeated invocations cycle through more matching choices. The default binding for `dabbrev-expand` is `M-/`.

As great as this is, I prefer `hippie-expand`, which is sort of like an extensible dabbrev, and in fact incorporates dabbrev as one of its expansion sources.

```
(global-set-key "\M-/" 'hippie-expand)
;; I want longer expansions (and especially try-expand-list) to come later
(setq hippie-expand-try-functions-list
      '(try-expand-all-abbrevs
try-expand-dabbrev
try-expand-dabbrev-all-buffers
try-expand-dabbrev-from-kill
try-complete-lisp-symbol-partially
try-complete-lisp-symbol
try-expand-line
try-expand-list
try-complete-file-name-partially
try-complete-file-name))
```

Printing

Setting up Emacs for printing is strangely complicated, but a lot of this complication comes from my choice of reverse-video. I define a fancy Hydra to make it easy to switch between the many ways to print.

```
;;; printing
(setq
 lpr-command "lpr"
 printer-name ""                        ;cups lpr blows up with any other value
 lpr-add-switches nil
 lpr-switches '("-U" "kdw1" "-o" "sides=two-sided-long-edge")
 ps-print-color-p t)
(defvar kw-print-command
  (format "%s %s" lpr-command (mapconcat 'identity lpr-switches " "))
  "*Preferred command to print a file, duplex.")
;; I run emacs with "reverse video"; if color-printing, the colors come out
;; unreadable on white paper; this advice flips to "normal video"
;; while printing...
(mapc
 (lambda (func) (advice-add func :around #'kw-with-print-colors))
```

```
 '(ps-print-buffer-with-faces
   ps-print-region-with-faces
   ps-spool-buffer-with-faces
   ps-spool-region-with-faces))

(defun kw-with-print-colors (f &rest args)
  "Evaluate (apply F ARGS) after switching the frame colors to \"normal video\", i.e.
black on white.  The original frame colors are guaranteed to be
restored after F returns.  This is typically used when generating
printed output from a colorized emacs buffer."
  (let ((fg (frame-parameter nil 'foreground-color))
(bg (frame-parameter nil 'background-color)))
    (unwind-protect
(progn
  (set-foreground-color "black")
  (set-background-color "white")
  (apply f args))
      (set-foreground-color fg)
      (set-background-color bg))))
(defun kw-print-buffer-by-file (&optional simplex)
  "Print the current buffer's file duplex by default; an argument means simplex.
This function assumes that your printer can handle this buffer's
file type.  The command used is constructed from `lpr-command'
and `lpr-switches'.  Duplex switches assume CUPS."
  (interactive "P")
  (let ((fn (buffer-file-name)))
    (unless fn (error "No file associated with %S" (current-buffer)))
    (apply
     '(call-process
       ,lpr-command nil nil nil
       ,@lpr-switches ,@(if (not simplex) '("-o" "sides=two-sided-long-edge")) ,fn))))

(with-demoted-errors "%s"
  (let ((key "<print>"))
    (require 'hydra)
    (global-set-key
     (kbd key)
     (condition-case err
 (defhydra printing-menu (:color blue :hint nil)
   "
^File^         ^Buffer^             ^Region^           ^Screenshot^
^^^^^^-------------------------------------------------------------------
_d_: duplex    _b_: colorized       _r_: colorized     _w_: window
_s_: simplex   _B_: no colors       _R_: no colors     _n_: window, no modeline
```

```
^ ^            ^ ^                  _f_: frame
"
   ("d" kw-print-buffer-by-file)
   ("s" (lambda () (interactive) (kw-print-buffer-by-file 1)))
   ("b" ps-print-buffer-with-faces)
   ("B" ps-print-buffer)
   ("r" ps-print-region-with-faces)
   ("R" ps-print-region)
   ("w" emacsshot-snap-window)
   ("n" emacsshot-snap-window-exclude-modeline)
   ("f" emacsshot-snap-frame)
   ("q" nil "cancel"))
       (error
(message "warning: %s unbound: %S" key err)
nil)))))
```

Shells and Processes

Asynchronous commands spawned via `M-&` tend to stick around for a long time. if you give a second `M-&`, Emacs will go through an interactive dialog about whether or not you really want to run another command. We turn that off. Additionally, most async commands don't generate any output (because they're intended to run in the background), so we turn off the initial pop-up of the async buffer (it'll pop up if the process generates any output.)

```
;; yes I really want to run that 2nd process
(setq async-shell-command-buffer 'new-buffer)
;; async buffer is just blank 99% of the time...
(setq async-shell-command-display-buffer nil)
```

`goto-address-mode` makes URLs and email addresses clickable with `C-c RET`.

```
(dolist (hook '(shell-mode-hook eshell-mode-hook))
  (add-hook hook #'goto-address-mode))
;; TODO bind <tab> to a function that uses completion-at-point if we're at a prompt,
;;      and kw-goto-address-next-link if not
(with-eval-after-load 'shell
  (dolist (key (list (kbd "C-c j") (kbd "<backtab>")))
    (define-key shell-mode-map key #'kw-goto-previous-address)))
;; eshell does weird things with eshell-mode-map!
;; (with-eval-after-load 'eshell
;;   (dolist (key (list (kbd "C-c j") (kbd "<backtab>")))
;;     (define-key eshell-mode-map key #'kw-goto-previous-address)))
```

`shell-mode` inherits from `comint-mode` so we customize some Comint features:

```
;;; shell-mode
(add-hook 'shell-mode-hook
  (lambda ()
    (local-set-key (kbd "C-c C-o") #'kw-comint-delete-output)
    (setq comint-input-ring-file-name "~/.zsh_history")
    (comint-read-input-ring t)
    (setq-local comint-output-filter-functions
'(comint-truncate-buffer
  ansi-color-process-output
  comint-postoutput-scroll-to-bottom
  comint-watch-for-password-prompt))
    (setq-local comint-process-echoes t)))
```

`shell-mode` buffers really choke on super-long lines, like you might get by innocently curling to some web service. This fixed that problem:

```
(add-to-list 'comint-preoutput-filter-functions #'kw-comint-chunkify t)
```

`shell-mode` grabs `TAB` and does its own completion, so `zsh` never sees it. But `shell-mode`'s completion only does file names; `zsh` does everything (command options in particular). The best improvement for this situation I've found is the combination of `fish-completion` and `bash-completion` (personally, I think `zsh`'s completion would be even better if it were supported…).

```
;; shell-mode completion
(add-hook 'shell-mode-hook
  (lambda ()
    (when (and (executable-find "fish")
       (require 'fish-completion nil t))
      (fish-completion-mode +1)
      (when (and (executable-find "bash")
 (require 'bash-completion nil t))
(setq fish-completion-fallback-on-bash-p t)))))
```

So that `shell-mode` buffers don't eat up all of memory, I set the maximum number of lines to the size of *War and Peace*.

```
(setq comint-buffer-maximum-size 65336)
```

Documents Files

PDF Files

PDFs are handled by `doc-view-mode`. I generate a lot of PDFs from Org Mode and Lilypond documents; when I recompile a PDF, I want the buffer displaying it to automatically update.

```
(defvar kw-pdf-viewer "llpp")                 ; my preferred external viewer
(add-to-list 'revert-without-query "\\.pdf\\'") ; PDF files change on disk when remade
(add-hook 'doc-view-mode-hook #'auto-revert-mode) ;better yet, auto-revert them
;; page-break-lines-mode needs to display a newline after the ^L if there isn't one!
(add-hook 'doc-view--text-view-mode (lambda () (page-break-lines-mode -1)))

(with-eval-after-load 'doc-view
  (add-hook 'pdf-view-mode-hook 'auto-revert-mode)
  (defvar kw-doc-view-show-tooltip t
    "*If not nil, show 'doc-view-mode' search tooltip in minibuffer.")
  (defun kw-doc-view-show-tooltip (arg)
    "Use the minibuffer for 'doc-view-mode' search tooltips.
With a prefix arg, use a real GUI tooltip."
    (interactive "P")
    (tooltip-show (doc-view-current-info) (unless arg kw-doc-view-show-tooltip)))
  (define-key doc-view-mode-map (kbd "C-t") 'kw-doc-view-show-tooltip))
```

Dired

`dired` is Emacs' built-in file manager: its Windows Explorer, its Finder, its Norton Commander, if you will. `dired-x` adds some useful functions to `dired-mode`. By default, `dired` doesn't list dot files, but I want to see them more often than not (you can always toggle them off with `C-u s`). `dired-dwim-target` is incredibly useful in the very common "two-panel" file management scenario, when you are copying or moving files between two directories. Finally, we enable the `a` command, `dired-find-alternate-file`, which isn't dangerous, just supposedly confusing.

I'm a huge fan of the human-readable file sizes you get with `ls -h`, so I arrange that for both `dired` and for `find-dired` and `find-grep-dired`; for the latter, I specify `-E` as a grep option so that I get extended (`egrep`) regexps.

```
(global-set-key (kbd "C-x C-d") 'dired) ; replaces list-directory
(with-eval-after-load 'dired
  (load "dired-x"))
(add-hook 'dired-mode-hook 'turn-on-gnus-dired-mode) ; TODO move me to dot.gnus
(add-hook 'dired-mode-hook 'hl-line-mode)
```

```
;; (add-hook 'dired-mode-hook                           ; ^ shouldn't create a new buffer
;;           (define-key dired-mode-map (kbd "^")
;;             (lambda ()
;;               "Go up in the same buffer."
;;               (find-alternate-file "..")))))
(setq dired-listing-switches "-alh") ; I know what dot files are, and like human-readable file sizes
(setq dired-dwim-target t)          ; suggest other visible dired buffer
(put 'dired-find-alternate-file 'disabled nil) ; enable this very useful command
(setq wdired-allow-to-change-permissions t)
(setq dired-guess-shell-alist-user
      '((,(rx ".pdf" eos)
 ,kw-pdf-viewer
 ,(mapconcat #'identity (cons lpr-command lpr-switches) " "))
(,(rx ".doc" (? "x") eos) "libreoffice")))
(setq find-ls-option '("-exec ls -ldh {} +" . "-ldh"))
(if (member "FreeBSD" (ignore-errors (process-lines "uname" "-s")))
    (setq find-grep-options "-qsE")
  (when (member "Linux" (ignore-errors (process-lines "uname" "-s")))
    (setq find-grep-options "-qE")))
;; code below stolen from dired-diff
(defun kw-dired-ediff (file)
  "Compare file at point with FILE using `ediff-files'.
If called interactively, prompt for FILE.
If the mark is active in Transient Mark mode, use the file at the mark
as the default for FILE.  (That's the mark set by \\[set-mark-command],
not by Dired's \\[dired-mark] command.)
If the file at point has a backup file, use that as the default FILE.
If the file at point is a backup file, use its original, if that exists
and can be found.  Note that customizations of `backup-directory-alist'
and `make-backup-file-name-function' change where this function searches
for the backup file, and affect its ability to find the original of a
backup file.

FILE is the first argument given to the `diff' function.  The file at
point is the second argument given to `diff'."
  (interactive
   (progn
     (require 'dired-aux)
     (let* ((current (dired-get-filename t))
  ;; Get the latest existing backup file or its original.
  (oldf (if (backup-file-name-p current)
    (file-name-sans-versions current)
  (diff-latest-backup-file current)))
  ;; Get the file at the mark.
```

```
  (file-at-mark (if (and transient-mark-mode mark-active)
    (save-excursion (goto-char (mark t))
    (dired-get-filename t t))))
  (separate-dir (and oldf
     (not (equal (file-name-directory oldf)
 (dired-current-directory)))))
  (default-file (or file-at-mark
    ;; If the file with which to compare
    ;; doesn't exist, or we cannot intuit it,
    ;; we forget that name and don't show it
    ;; as the default, as an indication to the
    ;; user that she should type the file
    ;; name.
    (and (if (and oldf (file-readable-p oldf)) oldf)
 (if separate-dir
     oldf
   (file-name-nondirectory oldf)))))
  ;; Use it as default if it's not the same as the current file,
  ;; and the target dir is current or there is a default file.
  (default (if (and (not (equal default-file current))
    (or (equal (dired-dwim-target-directory)
       (dired-current-directory))
default-file))
       default-file))
  (target-dir (if default
  (if separate-dir
      (file-name-directory default)
    (dired-current-directory))
(dired-dwim-target-directory)))
  (defaults (dired-dwim-target-defaults (list current) target-dir)))
     (list
      (minibuffer-with-setup-hook
  (lambda ()
    (set (make-local-variable 'minibuffer-default-add-function) nil)
    (setq minibuffer-default defaults))
(read-file-name
 (format "Diff %s with%s: " current
 (if default (format " (default %s)" default) ""))
 target-dir default t))))))
  (let ((current (dired-get-filename t)))
    (when (or (equal (expand-file-name file)
     (expand-file-name current))
      (and (file-directory-p file)
   (equal (expand-file-name current file)
```

```
  (expand-file-name current))))
      (error "Attempt to compare the file to itself"))
    (if (and (backup-file-name-p current)
     (equal file (file-name-sans-versions current)))
(ediff-files current file)
      (ediff-files file current))))
(defun kw-dired-mark-backups ()
  "Mark all backup files (files ending with ~) for use in later commands. [KW]"
  (interactive)
  (dired-mark-files-regexp "~\\'"))
(eval-after-load 'dired
  '(progn
     (define-key dired-mode-map (kbd "C-M-f") #'dired-next-subdir)
     (define-key dired-mode-map (kbd "C-M-b") #'dired-prev-subdir)
     (define-key dired-mode-map "*~" #'kw-dired-mark-backups)
     (define-key dired-mode-map "="  #'kw-dired-ediff)))
(with-demoted-errors "%s" (diredfl-global-mode +1)) ;more color
```

I like this hack for `image-dired`.

```
(with-eval-after-load 'image-dired
  (defun kw-dired-thumbs-jump ()
    "Jump to the file corresponding to this thumbnail in a 'dired' buffer."
    (interactive)
    (dired-other-window (file-name-directory (image-dired-original-file-name))))
  (define-key image-dired-thumbnail-mode-map "j" #'kw-dired-thumbs-jump))
```

Proced

`proced` is Emacs's `htop`. These tweaks make it more responsive.

```
(setq proced-auto-update-flag t proced-auto-update-interval 2)
```

Parentheses

Balanced parentheses (actually, many pairs of characters that act like parens) are very important to the programmer. Here we make them more visible, and set up global bindings to insert them pair-wise.

Pair	Keystroke	Note
`()`	`M-(`	Emacs default binding
`[]`	`M-[`	
`{}`	`M-{`	we leave this as `backward-paragraph`
`”`	M-’	we steal this from `abbrev-prefix-mark`
`""`	M-"	

```
(show-paren-mode 1)                     ; more noticeable
(mapc (lambda (binding) (define-key global-map binding 'insert-pair))
      ;; note that M-' steals abbrev-prefix-mark, but I don't use abbrevs
      '(,(kbd "M-'") ,(kbd "M-\"") ,(kbd "M-[") ))
```

TODO *Web Browsing*

```
(define-key kw-handy-keymap (kbd "f")   ; browse-url-firefox
  (lambda ()
    "Browse url at point in firefox."
    (interactive)
    (let ((url (thing-at-point-url-at-point)))
      (if url (browse-url-firefox url) (browse-url-firefox (read-string "URL: "))))))
(defalias 'find-url 'browse-url-emacs)
;; url
(setq url-cookie-untrusted-urls '(".*")) ; cookies: generally a bad idea
;; browse-url
(setq browse-url-browser-function #'eww-browse-url) ; default: eww
(setq browse-url-secondary-browser-function 'browse-url-firefox)
;; shr
(setq shr-image-animate nil)            ; animated gifs too slow / annoying
(setq shr-use-fonts nil)                ; no proportional fonts please!
(setq shr-width 84)                     ; shr's fill-column, effectively
(setq shr-use-colors nil)               ; no background colors
;; eww
(with-eval-after-load 'eww
  (add-to-list 'clean-buffer-list-kill-never-regexps "^\\*eww")
  (defun kw-eww-copy-link-url ()
    "Make the url at point the newest entry in the kill ring."
    (interactive)
    (let ((url (get-text-property (point) 'shr-url)))
      (message "%s" url)
      (kill-new url)))
  (defun eww-tag-body (cont)
    "KW: implementation of eww's own 'eww-tag-body' but without setting foreground
and background-color."                  ;
    (shr-generic cont))
  (add-hook 'eww-mode-hook
    (lambda ()
      (define-key eww-mode-map "k" #'kw-maybe-kill-buffer)
      (define-key eww-mode-map [(backtab)] #'shr-previous-link)
      (define-key eww-mode-map "[" #'kw-scroll-one-paragraph-back)
      (define-key eww-mode-map "]" #'kw-scroll-one-paragraph)
      (define-key eww-mode-map "}" #'forward-paragraph)
```

```
      (define-key eww-mode-map "{" #'backward-paragraph)
      (define-key eww-mode-map "y" #'kw-eww-copy-link-url)))
  (add-hook 'eww-after-render-hook
    (lambda ()
      (when (string-match "^https?://news.ansible.uk/" (eww-current-url))
(let ((msg "Fixing Ansible bullets...") buffer-read-only)
  (message msg)
  (save-excursion
    (goto-char (point-min))
    (while (search-forward "" nil t)
      (delete-char -1)
      (insert "•")))
  (message (format "%s done." msg)))))))
```

[0/1] Music and Audio

Audio Controls

```
(defun kw-pause-or-mute ()
  "Pause / unpause EMMS or (if not running) mute / unmute audio."
  (interactive)
  (require 'emms)
  (if emms-player-playing-p
      (emms-pause)                      ;pause or unpause
    (alsamixer-toggle-mute)))

(global-set-key [XF86AudioRaiseVolume] #'alsamixer-up-volume)
(global-set-key [kp-up]                #'alsamixer-up-volume)
(global-set-key [XF86AudioLowerVolume] #'alsamixer-down-volume)
(global-set-key [kp-down]              #'alsamixer-down-volume)
(global-set-key [XF86AudioMute]        'kw-pause-or-mute)
(global-set-key [kp-begin]             'kw-pause-or-mute)
(autoload 'emms-next "emms" "Start playing the next track in the EMMS playlist." t)
(global-set-key [kp-right]             'emms-next)
(autoload 'emms-previous "emms" "Start playing the previous track in the EMMS playlist." t)
(global-set-key [kp-left]              'emms-previous)
(autoload 'emms-start "emms" "Start playing the current EMMS track." t)
(global-set-key [XF86AudioStart]        'emms-start)
(autoload 'emms-stop "emms" "Stop any current EMMS playback." t)
(global-set-key [XF86AudioStop]        'emms-stop)
```

EMMS

```
(autoload 'emms "emms" nil t nil)
(autoload 'emms "emms-browser" nil t nil)
```

```
(with-eval-after-load 'emms
  (require 'emms-setup)
  (require 'emms-player-mpd)
  (emms-all)
  (add-to-list 'emms-info-functions 'emms-info-mpd)
  (setq emms-player-list '(emms-player-mpd))
  (setq emms-player-mpd-music-directory "~/mp3s")
  (require 'emms-volume)
  (setq emms-volume-change-function 'emms-volume-mpd-change)
  (setq emms-browser-mode-hook 'emms-cache-set-from-mpd-all)
  (ignore-errors
    (call-process "mpc" nil nil nil "consume" "off"))
  (define-key emms-browser-mode-map (kbd "SPC") #'emms-browser-toggle-subitems-recursively)
  (define-key kw-handy-keymap "m"
    (condition-case err
(progn
  (require 'hydra)
  (defhydra kw-emms (:hint nil :color blue)
    "\n_b_ browse _i_ info _p_ playlist _s_ stop  _t_ toggle "
    ("b" emms-smart-browse)
    ("i" emms-show)
    ("p" emms)
    ("s" emms-stop)
    ("t" emms-pause :color red)
    ("q" nil "exit")))
      (error
       (lambda () (error "failed to define kw-emms hydra"))))))
```

TODO *Remember*

```
(setq remember-handler-functions '(remember-append-to-file))
(setq remember-data-file (expand-file-name "~/notes/remember/Inbox.org"))
(setq remember-notes-initial-major-mode 'org-mode)
(setq remember-leader-text "* ")
```

TODO *Org Mode*

```
(put 'org-toggle-comment 'disabled t)
(setq org-replace-disputed-keys t)
(setq org-M-RET-may-split-line nil)
(setq org-special-ctrl-a/e 'reversed)
(setq org-refile-targets
      '((org-agenda-files :level . 1)
(nil :level . 1)))
```

```
(add-hook 'org-mode-hook 'org-indent-mode)
(add-hook 'org-mode-hook
  (lambda ()
    (with-demoted-errors "Ignoring: %S"
      (flyspell-mode 1))
    (setq indicate-empty-lines t
  tab-always-indent t
  show-trailing-whitespace t
  org-src-fontify-natively t
  org-catch-invisible-edits 'smart
  org-hide-emphasis-markers t)))

(with-eval-after-load 'org
  ;; for my fixed-pitch font
  (setq org-emphasis-alist (cons '("/" underline) org-emphasis-alist))
  ;; easy templates
  (require 'org-tempo)
  (add-to-list 'org-structure-template-alist
       '("se" . "src emacs-lisp"))
  (add-to-list 'org-structure-template-alist
       '("so" . "src ocaml"))
  (org-babel-do-load-languages
   'org-babel-load-languages
   '((emacs-lisp . t)
     (shell . t)
     (org . t)
     (ditaa . t)
     (ocaml . t)
     (gnuplot . t)
     (R . t))))

;; the agenda
(setq org-agenda-restore-windows-after-quit t)
(defvar kw-agenda-files-dir "~/notes/remember/")
(defun kw-buffer-is-rememberp (buf)
  "Is this buffer one of my \"remember\" buffers?
I.e. contained in the directory `kw-agenda-files-dir'?"
  (let ((fn (buffer-file-name (get-buffer buf)))
(dir (file-name-as-directory (expand-file-name kw-agenda-files-dir))))
    (and fn (string-match-p (rx bos (literal dir) any) fn))))
(when (file-exists-p (file-name-as-directory  kw-agenda-files-dir))
  (setq org-agenda-files `(,(file-name-as-directory  kw-agenda-files-dir))))
(setq org-agenda-include-diary t)
(setq org-agenda-custom-commands
```

```
      '(("b" todo "BUY")
("B" todo-tree "BUY")))
;; when adding a new appt with org-capture, arrange for notifications
(add-hook 'org-capture-after-finalize-hook #'org-agenda-to-appt)
;; run when starting Emacs and everyday at 12:05am
(org-agenda-to-appt)
(run-at-time "12:05am" (* 24 3600) #'org-agenda-to-appt)

;; global org commands and org additions to non-org variables
(global-set-key (kbd "C-c a") #'org-agenda)
(global-set-key (kbd "C-c l") #'org-store-link)
;; org leaves these files behind if you use dot, gnuplot, or ditaa src blocks
(with-eval-after-load 'dired
  (setq dired-garbage-files-regexp
(concat dired-garbage-files-regexp "\\|" (rx bos (or "gnuplot-" "babel-" "dot-" "ditaa-")))))
(dolist (ext '(".ilg" ".ind"))          ; add some latex extensions
  (add-to-list 'completion-ignored-extensions ext))

;; publishing

;; tufte-handout class for writing classy handouts and papers
;; select with #+LaTeX_CLASS: tuftehandout
(with-eval-after-load 'ox-latex
  (add-to-list 'org-latex-classes
       '("tuftehandout"
 "\\documentclass{tufte-handout}
\\usepackage{color}
\\usepackage{amssymb}
\\usepackage{amsmath}
\\usepackage{gensymb}"
 ("\\section{%s}" . "\\section*{%s}")
 ("\\subsection{%s}" . "\\subsection*{%s}")
 ("\\paragraph{%s}" . "\\paragraph*{%s}")
 ("\\subparagraph{%s}" . "\\subparagraph*{%s}")))
  ;; select with #+LaTeX_CLASS: tuftehandout
  (add-to-list 'org-latex-classes
       '("tuftebook"
 "\\documentclass{tufte-book}\n
\\usepackage{color}
\\usepackage{amssymb}
\\usepackage{gensymb}
\\usepackage{nicefrac}
\\usepackage{units}"
 ("\\section{%s}" . "\\section*{%s}")
```

```
 ("\\subsection{%s}" . "\\subsection*{%s}")
 ("\\paragraph{%s}" . "\\paragraph*{%s}")
 ("\\subparagraph{%s}" . "\\subparagraph*{%s}"))))
```

Exporting to LaTeX

```
;; doesn't work...
(ignore
 '(let ((rx (rx "-%latex")))
    (if (string-match-p rx (car org-latex-pdf-process))
(progn
  (setf (car org-latex-pdf-process)
(replace-regexp-in-string rx "-pdflatex='pdflatex --shell-escape'" (car org-latex-pdf-process)))
  (setq org-latex-listings 'minted))
      (warn "can't find %s in org-latex-pdf-process; no colorized source code!" rx))))
```

Org Present

Present "powerpoints" direct from Org.

```
(add-hook 'org-present-mode-hook
  (lambda ()
    (org-present-big)
    (org-display-inline-images)
    (org-present-read-only)
    (setq-local saved-mode-line-format mode-line-format)
    (setq mode-line-format nil)
    (toggle-frame-fullscreen)))

(add-hook 'org-present-mode-quit-hook
  (lambda ()
    (org-present-small)
    (org-remove-inline-images)
    (with-suppressed-warnings
(setq mode-line-format saved-mode-line-format))
    (org-present-read-write)))
```

Org Capture

```
(global-set-key (kbd "C-c r") #'org-capture)
(setq org-capture-templates
      `(("n" "Note" entry (file ,(concat kw-agenda-files-dir "Inbox.org"))
 "* %? %^G" :empty-lines-after 1)
("a" "Appointment" entry (file+headline ,(concat kw-agenda-files-dir "agenda.org") "Calendar")
 "* APPT %?\n%^t" :empty-lines-after 1)
```

```
("t" "Todo" entry (file+headline ,(concat kw-agenda-files-dir "agenda.org") "TODOs")
 "* TODO %?\n%a\n%i" :empty-lines-after 1)
("d" "Due Date" entry (file+headline ,(concat kw-agenda-files-dir "agenda.org") "Calendar")
 "* APPT %? :due:\nSCHEDULED: %t DEADLINE: %^t":empty-lines-after 1)
("T" "Travel Idea" entry (file+headline ,(concat kw-agenda-files-dir "travel.org") "Calendar")
 "* %? %^g" :kill-buffer :prepend :empty-lines-after 1)
("p" "Personal Idea" entry (file+headline ,(concat kw-agenda-files-dir "personal.org") "Ideas")
 "* %? %^g\n%u\n%a%i" :empty-lines-after 1)
("w" "Work Idea" entry (file+headline ,(concat kw-agenda-files-dir "work-someday.org") "Ideas")
 "* TODO %? %^g\n%u\n%a%i" :empty-lines-after 1)
("c" "Clipboard" entry (file ,(concat kw-agenda-files-dir "Inbox.org"))
 "* %?\n%x" :empty-lines-after 1)
("j" "Journal Entry" entry (file+olp+datetree ,(concat kw-agenda-files-dir "journal.org"))
 "* %? %^g" :empty-lines-after 1)
("u" "URL" entry (file+headline ,(concat kw-agenda-files-dir "Inbox.org"))
 "* %x :web:" :immediate-finish t :empty-lines-after 1)))
(setq gnus-icalendar-org-capture-headline '("Calendar"))
(with-eval-after-load 'gnus-icalendar
  (gnus-icalendar-org-setup))
```

TODO *Email: Gnus and Message Mode*

This needs a *lot* of work!

Message

```
(global-set-key [remap compose-mail] #'gnus-msg-mail)

(defun kw-gnus-keybinding (&optional arg quiet)
  "A function for C-x r to inc new mail."
  (interactive "P")
  (if (and (functionp 'gnus-alive-p)
   (gnus-alive-p))
      (if quiet
  (progn
    (gnus-group-get-new-news arg)
    (gnus-group-save-newsrc))
(switch-to-buffer "*Group*")
(gnus-group-get-new-news arg))
    (gnus)))

(global-set-key "\C-xR" 'mailpeek)

(defun message-goto-gcc ()
```

```
  "Move point to the Gcc header."
  (interactive)
  (message-position-on-field "Gcc" "To" "Newsgroups"))

(add-hook 'message-mode-hook
  (lambda ()
    (footnote-mode 1)
    (setq message-forward-as-mime t) ; default changed in v27.1
    (local-set-key [(control c) (control f) (g)] 'message-goto-gcc)
    (setq indicate-empty-lines t
  show-trailing-whitespace t)))
(setq message-generate-headers-first t
      message-send-mail-partially-limit 8000000
      message-required-mail-headers '(From Subject Date
   (optional . In-Reply-To)
   Message-ID
   (optional . User-Agent))
      message-header-setup-hook '(lambda ()
   (if (fboundp 'highlight-headers)
       (highlight-headers (point-min) (point-max) nil)))
      ;; default message-expand-name prefers EUDC if it has been used in this session!  way too many hits
      ;; see eudc-server-hotlist but setting bbdb first results in errors; see
      ;; [[file:~/notes/remember/emacs.org::*do eudc-query-form for just name seaton][do eudc-query-form f
      bbdb-complete-mail-allow-cycling t
      message-completion-alist '(("^\\([^ :]*-\\)?\\(To\\|B?Cc\\|From\\):" . bbdb-complete-mail))
      message-fcc-handler-function 'message-output)

(defvar kw-message-display-recipients-fields
  '("To" "Cc" "Bcc" "Fcc" "Gcc")
  "*List of RFC-822 mail header names which represent the recipients of a message.
Don't use ':'.")

(defun kw-message-display-recipients ()
  (cl-flet ((fmt-field (field)
   (let ((value (mail-fetch-field field)))
     (if value
 (format "%s: %s" field value)
       nil))))
    (let ((str (unwind-protect
 (progn
   (message-narrow-to-head)
   (mapconcat
    #'identity
    (filter (lambda (x) (not (null x)))
```

```
    (mapcar (function fmt-field)
    kw-message-display-recipients-fields))
    "\n"))
 (widen))))
      ;; TODO MODERNIZE
      (kw-minimal-popup-window str " Recipients" nil nil
(lambda (win buf config)
  (setq truncate-lines nil))))))

(defun kw-message-assert-reasonable-message ()
  "Raise an error if there are problems with this message."
  (let ((subj (mail-fetch-field "Subject")))
    (if (string-match "Fwd: Digested Articles" subj)
(progn
  (message-goto-subject)
  (error "Fix your Subject!")))))

(add-hook 'message-send-hook
  (lambda ()
    (let ((p (point))
  (buf (current-buffer))
  (wc (current-window-configuration)))
      (kw-message-assert-reasonable-message)
      (undo-boundary)
      (delete-trailing-whitespace)
      (kw-message-display-recipients)
      (unwind-protect
  (unless (y-or-n-p "Really send message? ")
    (with-current-buffer buf
      (undo))
    (error "Never mind."))
(set-window-configuration wc)
(goto-char p)))))
```

Mailpeek

```
(global-set-key "\C-xR" 'mailpeek)
```

Gnus

```
(setq read-mail-command 'gnus)
```

TODO *Miscellaneous Major Modes*

See Programming for major modes for programming languages.

"When creating new buffers, use auto-mode-alist to automatically set the major mode."[1]

[1] K., Philip.

```
(setq-default major-mode
      (lambda ()
(unless buffer-file-name
  (let ((buffer-file-name (buffer-name)))
    (set-auto-mode)))))
```

File- and Directory-Specific Modes and Settings

One subdirectory in my notes contains ancient files from my beloved Palm Pilot (!), which are encoded as Windows-1252...

```
(modify-coding-system-alist
 'file
 (format "\\'%s/personal/pedit" (expand-file-name "~/notes"))
 'windows-1252)
```

Diff Mode and Ediff

Ediff, the Emacs subsystem for diffing and merging files, is an amazingly powerful feature (and infinitely superior to `vimdiff`) that I use on a daily basis. However, by default it wants to pop up multiple frames (top-level windows), which I hate: changing `ediff-window-setup-function` fixes that.

```
;; multiframe ediff interface is appalling
(setq ediff-window-setup-function 'ediff-setup-windows-plain)
```

Also, it starts out with windows split vertically (i.e. one on top of the other) which isn't what I usually want, so I change `ediff-split-window-function` (in Ediff, there's a keystroke (|) that toggles this layout, so it's easy to change back for the occasional circumstance when I want the default behavior).

```
(setq ediff-split-window-function 'split-window-horizontally)
```

The Ediff hooks `ediff-before-setup-hook`, `ediff-suspend-hook`, and `ediff-quit-hook` are configured to save and restore the window configuration before and after diffing.

```
(defvar kw-ediff-window-config nil "Window config before ediffing.")
(add-hook 'ediff-before-setup-hook
  (lambda ()
    (setq kw-ediff-window-config (current-window-configuration))))
(dolist (hook '(ediff-suspend-hook ediff-quit-hook))
```

```
  (add-hook hook
    (lambda ()
      (set-window-configuration kw-ediff-window-config))))
```

I set `ediff-cleanup-hook` to run `ediff-janitor` to possibly delete buffers that contain files that were loaded just for diffing.

```
(add-hook 'ediff-cleanup-hook (lambda () (ediff-janitor t nil)))
```

I frequently diff `org-mode` files, which I've arranged to start up folded, so that as you move from diff to diff you can't actually see the folded differences! `ediff-prepare-buffer-hook` solves this problem by unfolding the buffers.

```
(add-hook 'ediff-prepare-buffer-hook
  (lambda ()
    (when (eq major-mode 'org-mode)
      (outline-show-all))))
```

I really want to notice trailing whitespace in my diffs, so that I can go back and eliminate it.

```
(add-hook 'diff-mode-hook
  (lambda ()
    (define-key diff-mode-map (kbd "M-p") #'kw-diff-hunk-prev)
    (define-key diff-mode-map (kbd "p") #'kw-diff-hunk-prev)
    (define-key diff-mode-map (kbd "M-n") #'kw-diff-hunk-next)
    (define-key diff-mode-map (kbd "n") #'kw-diff-hunk-next)
    (setq show-trailing-whitespace t)))
```

One of many modes where I want long lines wrapped.

```
(add-hook 'log-view-mode 'visual-line-mode)
```

Finally, I like v27.1's new compact file headers, and some of Ediff's default colors don't work well with my dark theme.

```
(when (version<= "27.1" emacs-version) ;use new compact file headers
  (setq diff-font-lock-prettify t))
(custom-set-faces ; default even/odd ediff faces don't work with reverse-video
  '(ediff-even-diff-A ((t (:background "Dim Grey"))))
  '(ediff-odd-diff-A ((t (:background "Dim Grey"))))
  '(ediff-even-diff-B ((t (:background "Dim Grey"))))
  '(ediff-odd-diff-B ((t (:background "Dim Grey")))))
```

Text Mode

Many text-like modes run `text-mode-hook`, just as many programming language modes run `prog-mode-hook`. The file `~/quotes` is a text file with a slightly strange format.

```
(add-hook 'text-mode-hook
  '(lambda ()
     (auto-fill-mode 1)         ; auto-fill aka word wrap
     (setq indicate-empty-lines t ; avoid excess whitespace
   show-trailing-whitespace t)))
(dolist (quotes-file '( "~/quotes" "~/notes/quotes"))
  (add-to-list
   'auto-mode-alist
   (cons
    (concat "\\'" (expand-file-name quotes-file) "\\'")
    (lambda ()
      (text-mode)
      (setq-local paragraph-separate "^%%\\s-*$")
      (setq-local paragraph-start    "^%%\\s-*$")))))
```

Version Control

It can speed things up (especially on remote systems, via Tramp) to restrict the dozen supported version control systems to the ones I actually use.

```
(setq vc-handled-backends '(RCS Git Hg)) ; speeds thing up, especially with tramp
```

I implement Mercurial's `addremove` subcommand here and bind it to `a` in `vc-dir-mode`.

```
(defun kw-hg-addremove ()
  "Do hg addremove -v in 'default-directory'.
If 'major-mode' is 'vc-dir-mode', revert the buffer."
  (interactive)
  (call-process "hg" nil nil nil "addremove" "-v")
  (when (derived-mode-p 'vc-dir-mode)
    (revert-buffer)))
(with-eval-after-load 'vc-dir
  (define-key vc-dir-mode-map "a" #'kw-hg-addremove))
```

In `vc-hg-log-view-mode` buffers, `C-c C-c` does my fancy Make-based `hg push` or `pull`; if no Makefile, fall back to the VC commands.

```
(defun kw-vc-log-view-push-pull ()
  "Do a 'vc-push' or 'vc-pull', as appropriate."
  (interactive)
  (cond
   ((equal (buffer-name) "*vc-incoming*")
    (if (eq 0 (call-process "make" nil nil nil "-n" "pull"))
(compile "make pull")
```

```
      (vc-pull '(4))))
   ((equal (buffer-name) "*vc-outgoing*")
    (if (eq 0 (call-process "make" nil nil nil "-n" "push"))
(compile "make push")
      (vc-push '(4))))
   (t (user-error "yikes"))))
(with-eval-after-load 'vc-hg
  (define-key vc-hg-log-view-mode-map (kbd "C-c C-c") #'kw-vc-log-view-push-pull))
```

Handy key binding to kill the VC buffer:

```
(with-eval-after-load 'vc-dir
  (define-key vc-dir-mode-map (kbd "k") 'kw-kill-buffer-and-window))
```

Programming

Every programmer will want to customize Emacs for each of the programming languages they use frequently. Emacs comes with built-in support for many languages (Emacs Lisp, of course; Lisp; Scheme; Java; C and C++, Python, etc); other less common languages have 3rd party support via the Emacs Package Manager.

Customizations common to all programming modes can be effected via `prog-mode-hook`. I want the `TAB` key to insert spaces (rather than a combination of spaces and tabs) in all modes. I also want structural folding with `yafolding-mode`, and useless whitespace made annoyingly obvious.

```
(setq-default indent-tabs-mode nil) ; make tabs insert spaces by default
(add-hook 'prog-mode-hook 'yafolding-mode)
(add-hook 'prog-mode-hook 'goto-address-prog-mode)
(add-hook 'prog-mode-hook
  (lambda ()
    (setq indicate-empty-lines t
  show-trailing-whitespace t)))
```

Compilation

I want to see the active part of the compilation process.

```
(setq compilation-scroll-output t)
```

I want `C-c C-c` to compile a file or project in programming language modes.

```
(with-eval-after-load 'prog-mode
  (define-key prog-mode-map [(control c) (control c)]
    (if (require 'projectile nil t)
```

```
#'projectile-compile-project
#'project-compile)))
```

Makefile Mode

I want `C-c C-c` in a Makefile to compile, as it does in most programming language modes, rather than `comment-region`.

```
(add-hook 'makefile-mode-hook
  #'(lambda ()
      (define-key makefile-mode-map [(control c) (control c)] #'compile)))
```

Emacs Lisp Programming

Prefer to load a `.el` file if it is newer than its `.elc`.

```
(setq load-prefer-newer t)
```

I want the `TAB` key to always (re)indent; `eldoc-mode` shows documentation for Emacs Lisp functions as you type them.

```
(add-hook 'emacs-lisp-mode-hook
  (lambda () (set (make-local-variable 'tab-always-indent) t)))
(add-hook 'emacs-lisp-mode-hook (lambda () (eldoc-mode 1)))
```

Short aliases for debugging are essential.

```
(defalias 'tdoe 'toggle-debug-on-error)
(defalias 'doe  'debug-on-entry)
(defalias 'cdoe 'cancel-debug-on-entry)
```

We want the debugger when we make an error in `eval-expression`.

```
(setq eval-expression-debug-on-error t)
```

Make `C-c C-c` do the nearest thing to the usual compile-this-file.

```
(add-hook 'emacs-lisp-mode-hook
  (lambda ()
    (define-key emacs-lisp-mode-map [(control c) (control c)] #'compile)))

;; can't live with the byte compiler's 80-column docstring restriction...
(setq byte-compile-warnings '(not docstrings))

(setq edebug-print-length 256)
```

Shell Programming

I hate aspects of `sh-mode`'s indentation.

```
(add-hook 'sh-mode-hook                 ; improve sh-mode indentation
  (lambda ()
    (setq sh-indent-for-case-label 0
  sh-indent-for-case-alt '+)))
```

Tcl Programming

I use `tclkit` as my Tcl interpreter.

```
(add-hook 'tcl-mode-hook
  (lambda () (setq tcl-application "tclkit")))
```

OCaml Programming

My primary programming language is Ocaml; I use `tuareg-mode` and many other support packages.

```
;; ocaml programming
```

OPAM Setup Extend the `load-path` so we can load Elisp from OPAM packages, and extend the `exec-path` so we can use applications (e.g. `merlin`) installed via OPAM.

```
;; check for API change or other warnings from opam!
(when (executable-find "opam")
  (let ((tmpfile (make-temp-file "minimacs")))
    (unwind-protect
(progn
  (call-process "opam" nil (list nil tmpfile) nil "var" "share")
  (let ((errs (with-temp-buffer (insert-file-contents-literally tmpfile) (buffer-string))))
    (unless (string-empty-p errs)
      (display-warning 'initfile errs))))
      (delete-file tmpfile)))
  (let ((opam-share (car (with-demoted-errors "%S" (process-lines "opam" "var" "share"))))
(opam-bin   (car (with-demoted-errors "%S" (process-lines "opam" "var" "bin")))))
    (when (and opam-share (file-directory-p opam-share))
      (add-to-list 'load-path (expand-file-name "emacs/site-lisp" opam-share)))
    (when (and opam-bin (file-directory-p opam-bin))
      (add-to-list 'exec-path opam-bin))))
```

Tuareg Mode

```
(add-to-list 'auto-mode-alist
     '(,(rx "/." (| "ocaml" "utop") "init" eos) . tuareg-mode))
```

Turn on `merlin-mode` and `ocp-index-mode` when we're editing OCaml code.

```
(dolist (hook '(tuareg-mode-hook caml-mode-hook))
  (add-hook hook (lambda () (with-demoted-errors "%s" (merlin-mode))) t)
  (add-hook hook (lambda () (with-demoted-errors "%s" (merlin-eldoc-setup))) t)
  (add-hook hook (lambda () (with-demoted-errors "%s" (ocp-index-mode)))))
```

Thanks to the otherwise wonderful Dune build system, I'm now forced to structure even the simplest projects into annoying subdirectories.

```
(add-hook 'tuareg-mode-hook
  (lambda ()
    (define-key tuareg-mode-map [(control c) (control c)]
      (if (require 'projectile nil t)
  #'projectile-compile-project
#'project-compile))))
```

When I do `C-x C-s` to get an OCaml REPL, I want a different command for Dune projects as opposed to older non-Dune projects. The old-school `ocaml` top-level doesn't work with Dune + the latest `tuareg-mode`, so I need to configure the hot new top-level, `utop`, and arrange to use it for Dune-based projects.

```
(autoload 'utop "utop" "Toplevel for OCaml" t)
(defvar kw-utop-init-file-name ".utopinit")
(setq utop-command (format "opam exec -- dune utop . -- -emacs -init %s" kw-utop-init-file-name))
(autoload 'utop-minor-mode "utop" "Minor mode for utop" t)
(add-hook 'tuareg-mode-hook 'utop-minor-mode)
(with-eval-after-load 'utop-minor-mode
  (dolist (k '("M-<return>" "S-<return>" "C-<return>"))
    (define-key utop-mode-map (kbd k) 'utop-eval-input-auto-end))
  (define-key utop-mode-map (kbd "C-c C-d") 'utop-kill))


(add-hook 'tuareg-mode-hook
  (lambda ()
    (if (locate-dominating-file default-directory "dune-project")
(setq-local tuareg-interactive-program utop-command)
      (setq-local tuareg-interactive-program "opam exec -- ocaml -nopromptcont"))))
```

This setting gives me multiple top-levels in one Emacs: one top-level per project.

```
(setq tuareg-interactive-buffer-name-generator
      (lambda ()
(tuareg-interactive-buffer-name-per-project ".hg" ".")))

(setq tuareg-match-patterns-aligned t)  ; align pipes in pattern matches
```

Here I define key bindings for some custom Tuareg extensions. I like to keep my source aligned vertically in certain constructs. See keybindings in Packages.

```
(defun kw-tuareg-align (str)
  "Align this group of lines on STR."
```

```
  (interactive)
  (let ((ln (line-number-at-pos))
minln maxln)
    (save-excursion
      (save-match-data
(beginning-of-line)
(when (not (looking-at (concat ".*" str)))
  (error "no %s here" str))))
    (save-excursion
      (save-match-data
(setq minln ln)
(beginning-of-line)
(while (and (re-search-backward str nil t) (= (line-number-at-pos) (- minln 1)))
  (cl-decf minln))))
    (save-excursion
      (save-match-data
(setq maxln ln)
(end-of-line)
(while (and (re-search-forward str nil t) (= (line-number-at-pos) (+ maxln 1)))
  (cl-incf maxln))))
    (align-regexp
     (save-excursion (progn (goto-char (point-min)) (forward-line minln)) (beginning-of-line) (point))
     (save-excursion (progn (goto-char (point-min)) (forward-line (+ 1 maxln))) (beginning-of-line) (point
     (concat "\\(\\s-*\\)" str))))
(defun kw-tuareg-align-> () (interactive) (kw-tuareg-align "->"))
(defun kw-tuareg-align-: () (interactive) (kw-tuareg-align ":"))
(defun kw-tuareg-assert-false ()
  "Insert 'assert false' at point."
  (interactive)
  (insert "assert false"))
```

Dune Mode

```
(autoload 'dune-mode "dune" "Major mode to edit dune files." t nil)
```

OCaml Interfacing with Other Subsystems My many OCaml source code directories contain `_build` directories that generally shouldn't be grepped..

```
(with-eval-after-load 'grep
  (add-to-list 'grep-find-ignored-directories "_build")) ;ocaml build artifacts)
```

TINT Programming

TINT is my version of the TRAC programming language (1959). I use it as an embedded language and for macro processing. My implementation includes a major and a minor mode. I need these autoloads because I install my `tint.el` via the OCaml package manager, rather than via the Emacs package manager.

```
(dolist (f '(run-tint tint-mode tint-eval tint-eval-at-point))
  (autoload f "tint" nil t))
```

EUDC—Emacs Unified Directory Client (LDAP)

I use EUDC to communicate with the UofC LDAP server. The configuration needs some tweaks for local strangenesses.

```
;; eudc aka ldap
(customize-set-variable 'eudc-server "ldaps://ldap.uchicago.edu")
(setq eudc-server-hotlist '((,eudc-server . ldap)))
(customize-set-variable 'eudc-default-return-attributes
'(displayname ucdepartment unit title uid mail mailLocalAddress gecos))
(customize-set-variable 'eudc-query-form-attributes '(name firstname mailLocalAddress uid))
(customize-set-variable 'ldap-host-parameters-alist
'(("ldaps://ldap.uchicago.edu"
   base "dc=uchicago,dc=edu"
   binddn "uid=dldcservice,ou=people,dc=uchicago,dc=edu"
   auth-source t)))
(with-eval-after-load 'eudc
  (eudc-set-server eudc-server 'ldap t)
  (add-to-list 'eudc-user-attribute-names-alist
       '(mailLocalAddress . "E-Mail")))
(defalias 'ldap 'eudc-query-form)
```

There's a bug in EUDC that manifests with the UofC LDAP server. This version of the function works for me. I need to submit this as a bug to Emacs!

```
(with-eval-after-load 'eudcb-ldap
  (defun eudc-bob-make-button (label keymap &optional menu plist)
    "Create a button with LABEL.
Attach KEYMAP, MENU and properties from PLIST to a new overlay covering
LABEL.  BUGFIX by KW."
    (let (overlay
  (p (point))
  prop val)
      (insert (or label "NONE"))
```

```
      (put-text-property p (point) 'face 'bold)
      (setq overlay (make-overlay p (point)))
      (overlay-put overlay 'mouse-face 'highlight)
      (overlay-put overlay 'keymap keymap)
      (overlay-put overlay 'local-map keymap)
      (overlay-put overlay 'menu menu)
      (while plist
(setq prop (car plist)
      plist (cdr plist)
      val (car plist)
      plist (cdr plist))
(overlay-put overlay prop val)))))
```

DLDC Web Publishing

We use my hacked version of Org Project exporting as a static site generator for our local documentation. Nothing to see here.

```
;; DLDC web publishing
(when (string= "jfcl" (system-name))
    (with-demoted-errors "%S"
      (load-file "/data/web/dldc/local/CONFIG/init.el")))
```

Games and Amusements

I use this big Scrabble dictionary for my Spelling Bee program.

```
(setq spelling-bee-dictionary-file
      (expand-file-name "~/docs/word-lists/twl06.txt"))
```

An occasional fortune is nice.

```
(setq fortune-dir "/usr/share/fortune/")
(setq fortune-always-compile nil)
(with-eval-after-load 'fortune
  (defun kw-switch-fortune-file (&rest ign)
    (unless current-prefix-arg
      (setq fortune-file
    (car
     ;; this dumb sort is standing in for a choose-random-list-element function
     (sort
      ;; fortune-mod's compiled files end in .dat...
      (directory-files fortune-dir t (rx bos (+ (not ?.)) eos))
      (lambda (_x _y) (= 1 (random 2)))))))))
  ;; switch fortune file before every fortune, unless prefix arg
  (dolist (func '(fortune fortune-message))
```

```
    (advice-add func :before #'kw-switch-fortune-file)))
```

The traditional TBIC function for `tetris`.

```
(with-eval-after-load 'tetris
  (define-key tetris-mode-map (kbd "<end>")
    (lambda ()
      "TBIC."
      (interactive)
      (let ((inhibit-message t))
(tetris-pause-game))
      (let ((buffer (current-buffer)))
(delete-windows-on buffer)
(bury-buffer buffer)))))
```

Arch Linux

I use my `pacman` package to run the Arch Linux `pacman(8)` command and do OS upgrades.

```
(use-package pacman :defer t
  :commands (pacman-mode pacman-Q pacman-Qi pacman-Ql pacman-Qo
 pacman-Qs pacman-Si pacman-Ss))
```

Recently after many (all?) updates, two of my X key mappings are unset, so I use this hook to restore them.

```
(add-hook 'pacman-after-Syu-hook
  (lambda ()
    (call-process "setxkbmap" nil nil nil
  "-option" "altwin:prtsc_rwin")
    (call-process "setxkbmap" nil nil nil
  "-layout" "us" "-option" "ctrl:nocaps")
    (message "Restored X key mappings.")))
```

Cocktail Database

Customization for my cocktail calculator package.

```
(defvar cocktail-sources
      (mapcar (lambda (it) (expand-file-name it "~/notes/cocktails/"))
      '("cocktails.db" "vintage.db" "baker.db")))
(defvar cocktail-history-source (expand-file-name "~/.cocktailcalc/history"))
(defvar cocktail-default-servings 2)
(add-hook 'cocktail-list-mode-hook #'hl-line-mode)
(add-hook 'cocktail-history-mode-hook #'hl-line-mode)
```

Project Skeletons

I use Skeletor for project boilerplate templates.

```
;; load skeletor skeletons
(ignore-errors
  (load-file (expand-file-name "~/emacs/project-skeletons/init.el")))
```

Packages

Here, finally, I load all the third-party packages (including my own) that I use. The `use-package` macro delays loading any of these until I actually use them.

```
(use-package tuareg
  :defer t
  :bind (:map tuareg-mode-map
      ("M-q"   . fill-paragraph)
      ("M-RET" . tuareg-interactive-send-input-end-of-phrase)
      ("C-c >" . kw-tuareg-align->)
      ("C-c :" . kw-tuareg-align-:)
      ("C-c A" . kw-tuareg-assert-false)))

;; my major mode for Chord files (lead sheets)
(use-package chord-mode :defer t
  :init
  (add-to-list 'auto-mode-alist
       (cons (format "%slyrics/" abbreviated-home-dir) 'chord-mode)))

;; major mode for editing CSV files
(use-package csv-mode :defer t
  :init (add-to-list 'auto-mode-alist '("\\.[Cc][Ss][Vv]\\'" . csv-mode)))

;; visible scrollbar replacement
(use-package sml-modeline
  :config (sml-modeline-mode 1))

;; the late lamented enhancement to Emacs pdf file handling (it still
;; works but Arch Linux's bleeding edge pdf libraries have
;; outdistanced this Emacs package...
(use-package pdf-tools :defer t
  :config
  (pdf-tools-install t nil t)
  (setq pdf-view-mode-hook
(lambda ()
```

```
  (with-demoted-errors "Ignoring: %S"
    (pdf-tools-enable-minor-modes)))))

;; minor mode for composing MIME articles; not sure if I still use this!
(use-package mml :defer t
  :config
  (defadvice mml-preview (after kw-mml-preview-advice)
    "Make mml-preview go full-screen."
    (delete-other-windows))
  (ad-activate 'mml-preview))

(use-package compile :defer t
  :config
  (add-to-list 'compilation-error-regexp-alist ; for refertool
       '("^%\\w+ +\\([^:\n]+\\):\\([0-9]+\\)" 1 2) t)
  (add-to-list 'compilation-error-regexp-alist    ;for qtest
       '("^Error: .*>\\(.*\\):\\([0-9]+\\)" 1 2)))

(use-package grep :defer t
  :config
  ;; from: refertool annotate -e
  ;; must append due to emacs bug or refertool annotate -e error!
  (add-to-list 'grep-regexp-alist '("^%\\w+ +\\([^:\n]+\\):\\([0-9]+\\)" 1 2) t))

;; google-translate
(use-package google-translate :defer t
  :config (require 'google-translate-smooth-ui))

  ;;; google-this
(use-package google-this
  :bind (("C-c /"   . google-this-mode-submap)
 :map google-this-mode-submap
 ("C-c / t" . google-translate-smooth-translate)
 ("C-c / /" . google-this)))

;;; page-break-lines
(use-package page-break-lines
  :init (global-page-break-lines-mode)
  :config (mapc (apply-partially 'add-to-list 'page-break-lines-modes)
'(text-mode org-mode tuareg-mode)))

;;; Nov-mode: read .epub files
(use-package nov :defer t
  :config
```

```
  (setq nov-text-width 84 nov-variable-pitch nil)
  (bind-keys :map nov-mode-map ("[" . kw-scroll-one-paragraph-back))
  (bind-keys :map nov-mode-map ("]" . kw-scroll-one-paragraph))
  :mode ("\\.epub\\'" . nov-mode))

;; can't load hyperbole without figuring out how to turn some of its
;; global key bindings off
;; (use-package hyperbole :bind (("C-=" . hui-select-thing)))

(use-package refer-mode :defer t
  :init
  (add-to-list 'auto-mode-alist '("\\.db\\'" . refer-mode))
  (add-hook 'refer-browse-mode-hook #'hl-line-mode)
  (add-hook 'refer-mode-hook #'goto-address-mode))

  ;;; docket
(use-package docket :defer t
  :init
  (dolist (mode '(prog-mode-hook org-mode-hook LilyPond-mode-hook))
    (add-hook mode #'docket-highlight-mode)))

;; quick
(use-package quick :defer t
  :bind (("<f12> q" . 'quick))
  :config
  (require 'midnight)
  (add-to-list 'clean-buffer-list-kill-never-buffer-names quick-buffer-name))

;; windsize
(use-package windsize
  :config (windsize-default-keybindings))

;; BBDB
(use-package bbdb :defer t
  :init (setq bbdb-file "~/emacs/bbdb")
  :config
  (setq bbdb-default-area-code 773
bbdb-default-country "USA"
bbdb/mail-auto-create-p nil
bbdb-use-pop-up nil
bbdb-completion-display-record nil
bbdb-completion-list '(aka fl-name lf-name mail)
bbdb-always-add-addresses 'never
bbdb-quiet-about-name-mismatches 0 ;was t
```

```
bbdb-complete-name-allow-cycling t
bbdb-offer-save nil)
  (bbdb-initialize 'gnus 'message)
  (bbdb-insinuate-gnus)
  :bind (:map bbdb-mode-map
      ("z" . quit-window)))

(use-package ocp-index :defer t
  :commands ocp-index-mode)

(use-package projectile
  :bind-keymap
  ("C-x p" . projectile-command-map)
  :config (setq projectile-completion-system 'default)
  (projectile-mode +1))

;; lilypond
(let ((debug-on-error nil))
  (with-demoted-errors "%S"
    (load-library "lilypond-init")
    (add-hook 'LilyPond-mode-hook
      (lambda ()
(with-demoted-errors "%S"
  (setf (cadr (assoc "LilyPond" LilyPond-command-alist))
;; the pkill is kind of extreme, but what does it hurt to reload any running llpp?
'(LilyPond-lilypond-command " %s && pkill -HUP llpp")))))))

(use-package acme :defer t
  :bind (("C-1" . acme-button-1)
 ("C-2" . acme-button-2)
 ("C-3" . acme-button-3)
 ("C-S-t" . acme-tag-jump-to-tag)))

(use-package move-text :defer t
  :bind (("M-<up>" . move-text-up)
 ("M-<down>" . move-text-down)))

;; fielded-mode
(use-package fielded :defer t
  :mode ("\\'/etc/\\(?:group-?\\|passwd-?\\|shadow-?\\)\\'" . fielded-mode))

;; nxml-mode
;; TODO do I still want to use nxml instead of html mode or web mode?
;; C-c C-c will view the html in eww
```

```
(use-package nxml-mode :defer t
  :init
  (mapc (lambda (ext) (add-to-list 'auto-mode-alist (cons ext 'nxml-mode)))
'("\\.xml$" "\\.html$" "\\.htm$" "\\.xhtml$" "\\.inc$" "\\.gema$" "\\.dgema$"))
  (setq nxml-sexp-element-flag t) ;sexp commands move over entire elements
  (add-hook 'nxml-mode-hook 'yafolding-mode)
  :config
  (bind-keys :map nxml-mode-map ("C-c C-c" . (lambda () (interactive) (eww-open-file (buffer-file-name))))

(use-package chess :defer t
  :config (setq chess-default-display 'chess-ics1))

(use-package empc :defer t
  :init (add-hook 'empc-mode-hook 'hl-line-mode))

(use-package dictionary :defer t
  :bind (("<f12> d" . 'dictionary-search)
 (:map dictionary-mode-map
       ( "[" . kw-scroll-one-paragraph-back)
       ( "]" . kw-scroll-one-paragraph))))

(use-package man :defer t
  :init (setq Man-width 82))

(use-package synosaurus
  :config (define-key kw-handy-keymap (kbd "s") synosaurus-command-map)
  :init
  (setq synosaurus-backend 'synosaurus-backend-wordnet)
  (setq synosaurus-choose-method 'ido))

(use-package flycheck :defer t
  :init
  (setq flycheck-standard-error-navigation nil) ; flycheck too aggressively takes over next-error!
  (setq flycheck-emacs-lisp-initialize-packages nil))

(use-package elfeed :defer t
  :init
  (setq elfeed-search-filter "@6-months-ago +unread ")
  (add-hook 'elfeed-show-mode-hook #'hl-line-mode)
  (add-hook 'elfeed-show-mode-hook
    (lambda ()
      (setq-local shr-max-image-proportion 0.4)))
  (add-hook 'elfeed-show-mode-hook
    (lambda ()
```

```
      (save-match-data
(save-excursion
  (goto-char (point-min))
  (when (re-search-forward "^Title:.*" nil t)
    (let ((inhibit-read-only t))
      (add-face-text-property (match-beginning 0) (match-end 0) '(:height 1.4))))))))
  (add-to-list 'clean-buffer-list-kill-never-buffer-names "*elfeed-entry*")
  :config
  (put 'elfeed-add-feed 'disabled t) ; i'm maintaining my feeds manually
  (load-file (concat (file-name-as-directory elfeed-db-directory) "kw-feeds.el"))
  (bind-keys :map elfeed-show-mode-map ("[" . kw-scroll-one-paragraph-back))
  (bind-keys :map elfeed-show-mode-map ("]" . kw-scroll-one-paragraph))
  (bind-keys :map elfeed-show-mode-map ("{" . backward-paragraph))
  (bind-keys :map elfeed-show-mode-map ("}" . forward-paragraph))
  (bind-keys :map elfeed-show-mode-map
     ("q" .
      (lambda ()
"Switch to *elfeed-search* buffer."
(interactive)
(switch-to-buffer "*elfeed-search*"))))
  (bind-keys :map elfeed-search-mode-map ("q" . elfeed-db-unload))
  (bind-keys :map elfeed-search-mode-map ("d" . elfeed-search-untag-all-unread))
  (bind-keys :map elfeed-search-mode-map ("SPC" . elfeed-search-show-entry)))

;; lorem-ipsum
(dolist (fn '(Lorem-ipsum-insert-sentences Lorem-ipsum-insert-paragraphs))
  (autoload fn "lorem-ipsum" nil t)
  (defalias (intern (downcase (symbol-name fn))) fn))
```

Handy Functions

what-face

```
;; Emanuel Berg <87zgrujcjm.fsf@zoho.eu>
(defun what-face (pos)
  (interactive "d")
  (let ((face (or (get-char-property pos 'face)
  (get-char-property pos 'read-cf-name) )))
    (message (format "%s" (or face "No face!"))) ))
```

kw-url-redirects

```
(defun kw-url-redirects (url)
  (interactive
   (list (read-string "URL: " nil nil (thing-at-point-url-at-point))))
```

```
  (require 'pcase)
  (require 'seq)
  (require 'thingatpt)
  (require 'cl-macs)
  (save-match-data
    (let* ((bname (format "*Redirects of %s*" (url-normalize-url url)))
   (buf (get-buffer-create bname))
   (url-request-method "HEAD")
   (my-map (make-sparse-keymap)))
      (with-current-buffer buf
(let ((inhibit-read-only t))
  (erase-buffer)
  (insert "   " url "\n")
  (special-mode)
  (set-keymap-parent my-map text-mode-map)
  (use-local-map my-map)
  (goto-address-mode +1)
  (setq-local kw-url-redirects-url url)
  (define-key my-map (kbd "<return>")
 (lambda (&optional _arg)
   "Invoke `browse-url-at-point'; if no URL at point, `browse-url' the starting URL."
   (interactive)
   (if (seq-some (lambda (o) (overlay-get o 'goto-address)) (overlays-at (point)))
       (browse-url-at-point)
     (browse-url kw-url-redirects-url))))
  (define-key my-map "k" 'kw-kill-buffer-and-window)
  (define-key my-map "q" 'quit-window)
  (cl-labels ((copy-url ()
"Copy the URL at point to the kill ring; if no URL at point, copy the starting URL."
(interactive)
(if (thing-at-point-url-at-point)
    (kill-new (thing-at-point-url-at-point))
  (kill-new kw-url-redirects-url)))
      (next-url ()
"Move to the next URL."
(interactive)
(let ((here (next-overlay-change (point))))
  (if (= (point-max) here)
      (user-error "End of buffer")
    (goto-char here)
    (unless (seq-some (lambda (o) (overlay-get o 'goto-address)) (overlays-at (point)))
      (goto-char (next-overlay-change (point)))))))
      (prev-url ()
"Move to the previous URL."
```

```
(interactive)
(let ((here (previous-overlay-change (point))))
  (if (= (point-min) here)
      (user-error "Beginning of buffer")
    (goto-char here)
    (unless (seq-some (lambda (o) (overlay-get o 'goto-address)) (overlays-at (point)))
      (goto-char (previous-overlay-change (point))))))))
    (define-key my-map (kbd "w") #'copy-url)
    (define-key my-map (kbd "<tab>") #'next-url)
    (define-key my-map (kbd "<backtab>") #'prev-url))))
      (url-retrieve
       url
       (lambda (status url buf)
 (let ((inhibit-read-only t)
       (prefix  "-> "))
   (with-current-buffer buf
     (dolist (pair (reverse (seq-partition status 2)))
       (pcase pair
 (`(:redirect ,u)
  (insert prefix u "\n"))
 (`(:error ,err)
  (insert prefix (propertize (format "%S\n" err) 'face 'error)))))
     (setq header-line-format
   (format "%d Redirects for %s"
   (length (seq-filter (apply-partially 'eq :redirect) status))
   url)))
   (kill-this-buffer))
 (pop-to-buffer buf))
       (list url buf) nil t))))
```

References

1. Knuth, Donald E. (1984). "Literate Programming". *The Computer Journal*. British Computer Society. 27 (2): 97–111. http://www.literateprogramming.com/knuthweb.pdf

2. K., Philp. "A Semi-literate Emacs Configuration". https://zge.us.to/emacs.d.html.