

Introduction This is a jekor’s `.emacs`¹, a configuration file for the Emacs editor.

I’ve been using Emacs for just over 2 years now. I started using it for SLIME, the “Superior Lisp Interaction Mode for Emacs”: a very powerful tool for Lisp programming and debugging. At the time I was a Vi/Vim user of 9 years. I didn’t intend to switch over to Emacs—let alone do so much inside of it—but I quickly fell in love.

What follows is a collection of various incantations to get Emacs to do all sorts of helpful or curious things. I’m publishing it in case it’s helpful to you and because I myself will probably forget exactly what I did years from now. I also wanted to experiment with the Noweb literate programming tool.

Enjoy.

```
1a <.emacs 1a>≡
    <Paths 1b>
    <List Functions 5a>
    <Appearance 3a>
    <Buffers and Files 7a>
    <Command Line 11c>
    <Information Management 12b>
    <Publishing 19b>
    <Mathematics 23c>
    <Programming 24a>
    <Automation 28a>
    <Internet 29a>
    <Music 31e>
    <Pictures 32b>
```

Filesystem Organization I keep all of my Emacs programs and runtime files—excluding this one—in a directory named `.emacs.d` within my home directory. This makes it easy to know what Emacs programs I’ve installed, to upgrade them, and to synch them between machines.

```
1b <Paths 1b>≡
    (defvar *emacs-path* "~/emacs.d")
    <Add to Path 2d>
    <Load Path 2a>
    <Info Path 2c>
    <Default Paths 2b>
    <Add to Paths 2e>
```

Defines:

`*emacs-path*`, used in chunks 2, 8, 9b, 27d, 28d, and 30d.

¹Copyright 2006–2009 Chris Forno

To add a new package, I move its directory into `~/emacs.d` (or create a directory for it if the package is a single Emacs file). I then add the directory containing the Emacs files for the package (if its not the root directory of the package) to Emacs's load path. This is very common, so there's a function for it.

2a `<Load Path 2a>≡`

```
(defun add-to-load-path (path &optional base)
  (add-to-path 'load-path path base))
```

Defines:

`add-to-load-path`, used in chunks [2](#), [5](#), [9e](#), [13a](#), [17-19](#), [22](#), [23](#), and [31e](#).

Uses `add-to-path` [2d](#).

Of course, some packages are too small to justify creating their own directory. Or sometimes I may want to write some 1-off Emacs code and place it directly into `~/emacs.d`. We can't use `add-to-load-path` for this or we'll get `~/emacs.d/~/emacs.d`.

2b `<Default Paths 2b>≡`

```
(add-to-list 'load-path *emacs-path*)
```

Uses `*emacs-path*` [1b](#).

A similar function exists for the less common task of adding a package's `.info` files to the info path. I rely on the implementation of `add-to-path` to prepend the path to the list of paths. This is important in the case of Info files so that the default Info directory lists are searched last.

2c `<Info Path 2c>≡`

```
(defun add-to-info-path (path &optional base)
  (add-to-path 'Info-default-directory-list path base))
```

Defines:

`add-to-info-path`, used in chunks [2e](#), [13a](#), [17d](#), [19c](#), and [22b](#).

Uses `add-to-path` [2d](#).

Both of the above are dependent on `add-to-path`: a helper for working with path lists.

2d `<Add to Path 2d>≡`

```
(defun add-to-path (type path &optional base)
  (let ((b (if base base *emacs-path*)))
    (add-to-list type (concat b "/" path))))
```

Defines:

`add-to-path`, used in chunk [2](#).

Uses `*emacs-path*` [1b](#).

Sometimes a directory will hold both the Emacs code and the Info documentation. Here's a shortcut.

2e `<Add to Paths 2e>≡`

```
(defun add-to-paths (path &optional base)
  (add-to-load-path path base)
  (add-to-info-path path base))
```

Defines:

`add-to-paths`, used in chunks [13d](#), [18b](#), [23d](#), and [26a](#).

Uses `add-to-info-path` [2c](#) and `add-to-load-path` [2a](#).

Appearance Since I spend so much time in Emacs, it's important that it's pleasing to look at. Not everyone (anyone?) will agree with my choices.

```
3a  <Appearance 3a>≡
      <Remove Clutter 3b>
      <Highlight Selection 3c>
      <Decorate Text 3d>
      <Hex Colors 4b>
      <Tone Down Parens 5b>
      <Color Theme 5c>
      <Indicators 5e>
      <Monitor Dimensions 6d>
      <Custom Faces 6e>
```

I reclaim as much screen space as possible and rely on the keyboard for all commands.

```
3b  <Remove Clutter 3b>≡
      (menu-bar-mode -1)
      (tool-bar-mode -1)
      (scroll-bar-mode -1)
      (setq inhibit-splash-screen 1
            inhibit-startup-message 1)
```

Almost everyone expects selected regions to be highlighted. It also helps to highlight the parenthesis at point along with its matching counterpart.

```
3c  <Highlight Selection 3c>≡
      (transient-mark-mode 1)
      (show-paren-mode 1)
```

I know of no situation where I don't want text styling. Enabling font-locking globally means that text will always be colored and styled when available.

```
3d  <Decorate Text 3d>≡
      (global-font-lock-mode 1)
      <Custom Fontlock Keywords 4a>
```

Some keywords like “TODO” and “FIXME” are very common and exist almost everywhere (especially in comments). I’d like for them to always stick out. To keep from highlighting other occurrences of “TODO” (such as in Org-Mode lists), I only highlight “TODO” when followed by a colon. “FIXME” is a different beast as it’s used as a placeholder and doesn’t usually make sense with a colon.

```
4a <Custom Fontlock Keywords 4a>≡
(defun add-custom-global-font-locking ()
  "Hilight some keywords globally."
  (interactive)
  (font-lock-add-keywords nil
    '(("\\<\\(FIXME\\)" 0 font-lock-warning-face t)
      ("\\<\\(TODO\\):" 0 font-lock-warning-face t)))
  (font-lock-mode 1))

(add-hook 'find-file-hook 'add-custom-global-font-locking)
```

One thing I’ve never seen in an editor is hex color highlighting (highlighting a hex color code with the color it represents). At first I thought I’d just use it in CSS, but I’ve noticed that I work a lot more with color than I originally realized. This makes dealing with colors a lot more intuitive and enjoyable.

This doesn’t catch all occurrences of color values, and it may catch some strings that aren’t colors, but for me it’s worth it. At some point I may add this to all modes by default (for now, I just add it to programming modes).

```
4b <Hex Colors 4b>≡
(defvar hexcolor-keywords
  '(("#[ABCDEFabcdef[:digit:]]\\{6\\}"
    (0 (put-text-property (match-beginning 0)
                          (match-end 0)
                          'face (list :background
                                      (match-string-no-properties 0)))))))

(defun hexcolor-add-to-font-lock ()
  (interactive)
  (font-lock-add-keywords nil hexcolor-keywords))

(add-to-hooks 'hexcolor-add-to-font-lock
  'css-mode-hook
  'php-mode-hook
  'html-mode-hook
  'elisp-mode-hook
  'haskell-mode-hook)
```

Uses add-to-hooks 5a.

From here on I often add some customization to a number of different modes. `add-to-hooks` makes the code less verbose.

```
5a <List Functions 5a>≡
    (defun add-to-hooks (action &rest hooks)
      (mapc #'(lambda (x)
                (add-hook x action)) hooks))
```

Defines:

`add-to-hooks`, used in chunks [4b](#), [5a](#), and [25b](#).

When working with Lisp in Emacs, the parentheses aren't important to focus on. Paren matching takes care of the details, and you can focus on indentation. Interestingly, after getting comfortable with Lisp and good parentheses matching, it seems I'm more comfortable with writing heavily-nested parenthetical expressions and find myself using paren matching in many other modes.

```
5b <Tone Down Parens 5b>≡
    (add-to-load-path "parenface")
    (require 'parenface)
```

Uses `add-to-load-path` [2a](#).

And here's the controversial bit that you probably won't agree with me on: a color theme. The color theme package is not included with Emacs.

```
5c <Color Theme 5c>≡
    (add-to-load-path "color-theme-6.6.0")
    (require 'color-theme)
```

Uses `add-to-load-path` [2a](#).

`color-theme-initialize` used to fail on one of my computers for some reason (although `color-theme` calls will still work), so ignoring the error seemed easiest.

```
5d <Color Theme 5c>+≡
    (ignore-errors (color-theme-initialize))
    (color-theme-dark-blue2)
```

Other good color themes are `charcoal-black`, `dark-blue2`, `hober`, `jsc-dark`, `ld-dark`, `oswald`, `pok-wob`, `pok-wog`, `sgml-*`, `shaman`, `simple-1`, `subtle-hacker`, `taming-mr-arneson`, `tty-dark`, and `word-perfect`.

Since I spend most of the time with Emacs fully maximized (I use `xmonad` without any bars or docks), it's useful to keep an eye on certain gauges and indicators. Other indicators, like column numbers, don't come from processes external to Emacs, but are useful when interacting with them (for example, compiler error messages).

```
5e <Indicators 5e>≡
    <Clock 6a>
    <Column Number 6b>
    <Visual Bell 6c>
```

Show the current time on the mode line.

```
6a <Clock 6a>≡
    (display-time-mode 1)
```

Column numbers are useful for Haskell error messages, among other things.

```
6b <Column Number 6b>≡
    (column-number-mode 1)
```

And I can't stand beeping. Let's change that to a visual indicator.

```
6c <Visual Bell 6c>≡
    (setq visible-bell t)
```

On some systems—namely OS X—the monitor dimensions are important. This is especially true for things like imaxima. The dimensions are the actual display surface in millimeters (432mm wide x 324mm tall).

```
6d <Monitor Dimensions 6d>≡
    ;; Samsung SyncMaster 204B - vertical
    (when (equal (system-name) "neo")
          (setq display-mm-dimensions-alist '((t . (432 . 324)))))
```

Finally, there are a number of colors and fonts I changed for various modes. I chose them from within a M-x `customize` so that I could preview how they would look. `default` is the default style of text and `secondary-selection` is how text is highlighted.

```
6e <Custom Faces 6e>≡
    (custom-set-faces
     '(default ((t (:stipple nil :background "#233B5A" :foreground "#fff8dc"
                  :inverse-video nil :box nil :strike-through nil
                  :overline nil :underline nil :slant normal :weight normal
                  :height 200 :width normal :family "apple-monaco"))))
     '(secondary-selection ((t (:background "cyan4"))))
     <Direc Faces 10d>
     <Org-Mode Faces 16c>))
```

Note that the above font setting will not work correctly without Xft on X11. To enable Xft, add `emacs.FontBackend: xft` to your `.Xdefaults` or `.Xresources` file.

Buffers and Files Emacs operates on buffers, which usually represent files in the filesystem. I tend to have a lot of buffers open at a time (usually around 100, but sometimes up to 300). Any improvement I make in the managing buffers results in greater editing power.

In fact, I consider buffers to be one of the most powerful features of Emacs. Other editors have them in one form or another, but they tend to be limited to what can be seen on screen (tabs).

```
7a <Buffers and Files 7a>≡
    <Uniquify Buffer Names 7b>
    <Tab Completion 7c>
    <Save Session 8b>
    <Disable Autosaves and Backups 9a>
    <Bookmarks 9b>
    <Searching for Files 9c>
    <Browsing Directories 9e>
    <Remote Filesystems 11a>
```

Sometimes buffers will have the same name. This happens often with common names like `README` as well as multiple versions of the same file. Emacs's default approach is to append `<n>` to the end of duplicate buffer names (where `n` increases with each duplicate).

An approach I find more informative is to append as much of the filesystem path as is necessary to disambiguate the buffer (`post-forward`). So instead of `README<2>` I get `README|someprogram`. This can be done with the `uniquify` package.

```
7b <Uniquify Buffer Names 7b>≡
    (require 'uniquify)
    (setq uniquify-buffer-name-style 'post-forward)
```

In order to quickly switch between buffers, it helps to be able to tab-complete their names. The `ido` package provides this and more. It displays the most recently used buffers in a neat 2-line display with `C-x b` and dynamically updates the list as you type. I actually use `<Return>` when I've found the buffer I want, but the name "tab completion" is already well established.

```
7c <Tab Completion 7c>≡
    (require 'ido)
    (ido-mode t)
```

It's rare for me to end an Emacs session (usually only while upgrading X or GNU/Linux), but in case I do, I like to keep track of what buffers I was using.

```
8a <Tab Completion 7c>+≡
    (setq ido-save-directory-list-file (concat *emacs-path* "/ido.last"))
```

Uses `*emacs-path*` [1b](#).

Of course, saving which buffers I was using most recently isn't any good unless I also save which buffers I had open. Emacs has built-in support for saving your "desktop" (session).

```
8b <Save Session 8b>≡
    (desktop-save-mode 1)
    (setq desktop-dirname *emacs-path*
          desktop-base-file-name "emacs.desktop")
```

Uses `*emacs-path*` [1b](#).

Some kinds of buffers I don't want to save between sessions. I personally won't care about directories I was browsing info files I was reading later. Also, I don't want to keep most configuration files open between sessions.

```
8c <Save Session 8b>+≡
    (setq desktop-buffers-not-to-save
          (concat "\\(" "^nn\\.a[0-9]+\\\\\\\\.log\\\\\\\\(ftp)\\\\\\\\|^tags\\\\\\\\|^TAGS"
                  "\\\\\\\\\.emacs.*\\\\\\\\\\\\.diary\\\\\\\\\\\\.newsrsrc-dribble\\\\\\\\\\\\.bbdb"
                  "\\\\\\\\\$"))
          (add-to-list 'desktop-modes-not-to-save 'dired-mode)
          (add-to-list 'desktop-modes-not-to-save 'Info-mode)
          (add-to-list 'desktop-modes-not-to-save 'info-lookup-mode)
```

I also save my command history and file name history.

```
8d <Save Session 8b>+≡
    (setq history-length 250)
    (add-to-list 'desktop-globals-to-save 'file-name-history)
```

And finally, I like to save where I was in each file. This is especially important for Org-Mode files, which can be thousands of lines long.

```
8e <Save Session 8b>+≡
    (require 'saveplace)
    (setq-default save-place t)
    (setq save-place-file (concat *emacs-path* "/emacs.places"))
```

Uses `*emacs-path*` [1b](#).

Emacs takes a paranoid approach to backing up files. While I normally appreciate paranoia, I disable autosaving (saving at regular intervals) and backups (retaining old versions of a file when saving). Although I have been burned a couple times by not having these, I think the burden outweighs the benefits.

```
9a <Disable Autosaves and Backups 9a>≡
    (setq auto-save-default nil)
    (setq make-backup-files nil)
```

Although I don't use bookmarks often, I want the bookmarks file in `~/ .emacs.d`.

```
9b <Bookmarks 9b>≡
    (setq bookmark-default-file (concat *emacs-path* "/emacs.bookmarks"))
```

Uses `*emacs-path*` 1b.

One day a colleague showed me a useful feature of TextMate. He could setup a “project” and then quickly search for files in that project with a keyboard shortcut. I figured that Emacs should have something similar, and sure enough it did. In my opinion, Emacs's approach is better, because it offloads the work to the `locate` command and allows you to search your entire filesystem. All you have to do is make sure your `locate` database is up-to-date.

```
9c <Searching for Files 9c>≡
    (require 'globalff)
    (global-set-key (kbd "C-x t") 'globalff)
```

Another handy trick is to quickly open the filename that's currently “at point” (under the cursor) in whatever file you're viewing.

```
9d <Searching for Files 9c>+≡
    (ffap-bindings)
```

On the subject of searching for files, I quite often search for text in the files of a directory. Emacs has the `rgrep` (recursive `grep`) command built-in. Just `M-x rgrep` and enjoy the highlighted and linked results that you receive. It also avoids recursing into any revision control directories automatically.

Browsing Directories Emacs has a built-in directory browser called `Dired`. It's similar to `Midnight Commander`. But there is an improved version not shipped with Emacs called `Dired+`.

```
9e <Browsing Directories 9e>≡
    (add-to-load-path "dired+")
    (require 'dired+)
    <Dired Faces 10d>
```

Uses `add-to-load-path` 2a.

Dired greys out files it thinks are uninteresting (backups, etc.). For some reason, it thinks that PDFs are uninteresting. Let's change that.

```
10a  <Browsing Directories 9e>+≡
      (setq dired-omit-extensions (delete ".pdf" dired-omit-extensions))
```

In the name of protecting the user, Dired will not recursively copy or delete directories without confirmation. Hey, I've used the command line and if I want to shoot myself in the foot, I will.

```
10b  <Browsing Directories 9e>+≡
      (setq dired-recursive-copies t
            dired-recursive-deletes 'always)
```

By default Dired will create new buffers every time you navigate into a new directory (by pressing <Return>). You can reuse the existing Dired buffer by pressing **a**, but only if you enable it.

```
10c  <Browsing Directories 9e>+≡
      (put 'dired-find-alternate-file 'disabled nil)
```

I think Dired could be a little bit more informative with color. I especially like to see file permissions at a glance.

```
10d  <Dired Faces 10d>≡
      '(diredp-compressed-file-suffix ((t (:foreground "red3"))))
      '(diredp-date-time ((t (:foreground "SpringGreen1"))))
      '(diredp-deletion-file-name ((t (:background "red2"))))
      '(diredp-dir-heading ((t (:background "black" :foreground "SpringGreen1"))))
      '(diredp-dir-priv ((t (:background "snow" :foreground "RoyalBlue4"))))
      '(diredp-display-msg ((t (:foreground "white"))))
      '(diredp-exec-priv ((t (:background "green"))))
      '(diredp-executable-tag ((t (:foreground "green"))))
      '(diredp-file-name ((t (:foreground "snow"))))
      '(diredp-file-suffix ((t (:foreground "MediumPurple1"))))
      '(diredp-flag-mark ((t (:background "SpringGreen2" :foreground "magenta"))))
      '(diredp-flag-mark-line ((t (:background "SpringGreen2"))))
      '(diredp-ignored-file-name ((t (:foreground "slate gray"))))
      '(diredp-link-priv ((t (:foreground "coral1"))))
      '(diredp-no-priv ((t nil)))
      '(diredp-rare-priv ((t (:background "orange1"))))
      '(diredp-read-priv ((t (:background "blue"))))
      '(diredp-symlink ((t (:foreground "coral1"))))
      '(diredp-write-priv ((t (:background "Red"))))
```

Remote Filesystems Often times I’ll want to edit files on a remote machine (especially when doing system administration work). I could create an SSH tunnel from the remote machine to my machine and use `emacsclient`, but very often Emacs isn’t installed on servers. Or I could create an SSH mount with `FUSE`, but that requires a lot of setup.

A quicker solution is to use the SSH method of Emacs’s TRAMP mode (Transparent Remote Access, Multiple Protocols). All I need to do to open a file on a remote machine is enter `/username@host:path` as the filename. This opens a buffer that I can work on as if it were local.

One problem I’ve run into is TRAMP hanging while waiting for a prompt from the remote host. For some reason, the default shell prompt regular expression is very restrictive.

```
11a <Remote Filesystems 11a>≡
      (setq tramp-default-method "ssh"
            tramp-shell-prompt-pattern "[^#>\n]*[#>]") *)
```

Just in case auto-save isn’t disabled already, disable it for TRAMP files unless you enjoy having Emacs hang every 5 minutes while you’re working on a remote file.

```
11b <Remote Filesystems 11a>+≡
      (remove-hook 'find-file-hook 'tramp-set-auto-save)
```

Command Line The first thing I do when I get in to X is to start Emacs. The actual command I use is: `emacs-23 --font "mono-20"`².

```
11c <Command Line 11c>≡
      <Emacs Server 11d>
      <Terminal Emulator ANSI Color 12a>
```

Once I’ve started Emacs, I let it run until my X session ends—usually months later. When I was still adjusting to Emacs, I still spent a lot of time at the command line. It was a real bother to switch from a terminal to Emacs to open a file there. A much nicer solution is typing `emacsclient file` to open the file. In order to do this, you need to have Emacs running in “server mode”.

```
11d <Emacs Server 11d>≡
      (require 'server)
      (server-start)
```

²I haven’t quite figured out how to get it working properly with just font commands in this file.

While I rarely use `emacsclient` on the command line anymore, I do have my `EDITOR` environment variable set to `emacsclient` so that when I use revision control programs from the command line a buffer pops up in Emacs asking me for the changeset description (also becoming rarer thanks to Emacs's built-in [version control](#)).

Note that Emacs in server mode by default only accepts local connections. You can set it to allow TCP connections as well. See the `server-use-tcp` and `server-host` variables. You may also want to look into using SSH tunnels if you need to use the server over an unsecure network.

Terminal Emulator Sometimes—like when I'm on OS X and don't have xmonad—it's nice to use Emacs's terminal emulator. You get a shell in a buffer.

ANSI color control codes aren't cheap to process, but for me they're worth it (and I don't notice any major slowdown).

```
12a <Terminal Emulator ANSI Color 12a>≡
    (require 'ansi-color)
    (autoload 'ansi-color-for-comint-mode-on "ansi-color" nil t)
    (add-hook 'shell-mode-hook 'ansi-color-for-comint-mode-on)
```

Information Management is what I call this next section for lack of a better term. I store nearly all of my information in text files. I don't use word processors, rarely use spreadsheets and databases, and never use programs with proprietary or complex formats for storing anything of importance.

```
12b <Information Management 12b>≡
    <Org-Mode 13a>
    <Calendar 17a>
    <Contacts 17d>
    <Encryption 18b>
    <Accounting 18c>
```

Org-Mode I basically live in **Org-Mode**. That is to say, I keep almost all of my personal information in text files using Org-Mode. I use it for todos, notes, journal entries, check lists, wish lists—basically anything that can be sorted into a list or outline.

Org-Mode is included with recent Emacs, but I like to use the latest version.

```
13a <Org-Mode 13a>≡
      (add-to-load-path "org-6.17c/lisp")
      (add-to-info-path "org-6.17c/doc")
      (require 'org-install)
      <Set Org-Mode as Default 13b>
      <Org Completion 13c>
      <Wrap Long Lines 14c>
      <Org Remember 13d>
      <Task Management 14d>
      <Org-Mode Keybindings 16a>
      <Org-Mode Faces 16c>
```

Uses `add-to-info-path 2c` and `add-to-load-path 2a`.

I wasn't kidding about living in Org-Mode. I use it as a default mode for any buffers that don't have another mode. Yes, this means goodbye to fundamental mode.

```
13b <Set Org-Mode as Default 13b>≡
      (setq default-major-mode 'org-mode)
```

Since we're already using `ido`, why not have Org-Mode use it too?

```
13c <Org Completion 13c>≡
      (setq org-completion-use-ido t)
```

After working a lot with the “Getting Things Done” (**GTD**) system, I like to be able to quickly capture ideas as I have them. Org-Mode integrates well with the `remember` package to handle this. I call it up from anywhere I'm at with `C-M-r`.

```
13d <Org Remember 13d>≡
      (add-to-paths "remember-2.0")
      (require 'remember)
      (org-remember-insinuate)
      (global-set-key (kbd "C-M-r") 'remember)
```

Uses `add-to-paths 2e`.

Whenever I call up `remember`, I want to do 1 of 3 things:

1. Make a note about something I need to do.
2. Record an idea I just had.
3. Write a quick journal entry.

`Remember` can do this by using templates. Each template has the form *name keyboard-shortcut template file headline* where the headline is the Org-Mode headline under which to insert the new item.

```
14a <Org Remember 13d>+≡
      (setq org-remember-templates
            '(("Todo"      ?t "* TODO %?\n %i\n %a"   org-default-notes-file "Tasks")
              ("Idea"     ?i "* %^{Title}\n %i\n %a"   org-default-notes-file "New Ideas")
              ("Journal"  ?j "* %U %?\n\n %i\n %a"   org-default-notes-file)))
```

For the templates to work, they need to know my default notes file.

```
14b <Org Remember 13d>+≡
      (setq org-directory "~/document"
            org-default-notes-file (concat org-directory "/personal"))
```

Org-Mode has an annoying default behavior of truncating long lines. This might make sense if you were just using Org-Mode for simple outlines, but considering that I use it for nearly everything, it makes more sense to wrap the lines, even if they are headings.

```
14c <Wrap Long Lines 14c>≡
      (add-hook 'org-mode-hook 'toggle-truncate-lines)
```

Task Management I originally started using Org-Mode for project management. For a long time I had tried to make other systems work: a 3-ring binder, index cards (the “hipster”), and a PDA. The non-electronic versions left me frustrated, and the PDA locked my data into an inflexible proprietary format.

I eventually decided to give up on a mobile solution and gave Org-Mode a try (based on some great [articles](#) by Sacha Chua). If learning Lisp hadn’t gotten me to switch to Emacs, Org-Mode for project management would have.

I call this section task management because I find myself managing tasks more often than projects (or the projects I have are large and ongoing and aren’t helpful when they stick around in my files forever, especially because they evolve so rapidly).

```
14d <Task Management 14d>≡
      <Task States 15a>
      <Task Color Coding 15c>
      <Task Logging 15d>
```

Tasks move between different states. Most often they're in the TODO state (haven't been started). Sometimes I've started them and have paused for some reason (sleep, lunch, etc.). Other times I'm waiting for someone. I also like to see and archive completed or cancelled tasks.

```
15a <Task States 15a>≡
      (setq org-todo-keywords
        '(("type" "TODO(t)" "STARTED(s!)" "DEFERRED(d@/!)" "DELEGATED(g@/!)"
          "|" "DONE(f@/!)" "CANCELLED(c@/!)")))
```

The "|" represents a logical separation between “done” and “not done” states. Each @ means that the task should receive a note when switched to this state (see Task Logging). Each ! means that the task should receive a timestamp when switched.

To change the state of a task, I use C-c C-t (`org-todo`). I have too many states to cycle through them, so I use the fast (direct) selection method. Basically, once I've pressed C-c C-t I enter the character representing the state I want (the character in parenthesis in the keywords above).

```
15b <Task States 15a>+≡
      (setq org-use-fast-todo-selection t)
```

To scan a list quickly, it helps to have items color-coded. This is especially important because I don't spend the time to organize my lists; they are long and sprawling.

```
15c <Task Color Coding 15c>≡
      (setq org-todo-keyword-faces
        '(("TODO" . font-lock-warning-face)
          ("STARTED" . org-todo)
          ("DEFERRED" . shadow)
          ("DELEGATED" . org-warning)))
```

I think it'll be useful in the future to know what the resolution to a task was. I've not ever looked back yet, but I consider each completed task as something of a journal entry. Consider it “literate task management”, if you will. This also records a timestamp for when I completed the task (which is useful except for when I forget to close a task until a couple months later).

```
15d <Task Logging 15d>≡
      (setq org-log-done '(state done))
```

Org-Mode Keybindings Org-Mode is all about moving around quickly. Most keybindings are pre-configured, but we need a few more.

First, here are some keybindings that the Org-Mode manual will tell you are a must. They allow you to use common functions of Org-Mode in non-Org-Mode buffers.

```
16a <Org-Mode Keybindings 16a>≡
      (global-set-key (kbd "C-c l") 'org-store-link)
      (global-set-key (kbd "C-c a") 'org-agenda)
```

Org-Mode has a handy function `org-open-at-point` which opens a hyperlink at point (using the configured browser). It's bound to `C-c C-o`. It's nice to be able to use the same function other places.

```
16b <Org-Mode Keybindings 16a>+≡
      (global-set-key (kbd "C-c C-o") 'org-open-at-point-global)
      (global-set-key (kbd "C-c C-l") 'org-insert-link-global)
```

By default, Org-Mode's colors clash with my color theme.

```
16c <Org-Mode Faces 16c>≡
      '(org-done ((t (:foreground "SpringGreen" :weight bold))))
      '(org-hide (((background dark) (:foreground "darkblue"))))
      '(org-todo ((t (:foreground "firebrick" :weight bold))))
      '(org-warning ((t (:foreground "Orange" :weight bold))))
      '(org-level-1 ((t (:foreground "cyan" :weight bold))))
      '(org-level-2 ((t (:foreground "white" :weight bold))))
      '(org-level-3 ((t (:foreground "sky blue" :weight bold))))
      '(org-level-4 ((t (:foreground "cornflower blue" :weight bold))))
      '(org-level-5 ((nil (:foreground "cyan"))))
      '(org-level-6 ((nil (:foreground "white"))))
      '(org-level-7 (((class color) (min-colors 16) (background dark))
                     (:foreground "sky blue"))))
      '(org-level-8 ((nil (:foreground "cornflower blue"))))
```

Calendar Apart from journal entries and tasks, I like to keep track of events and appointments. Org-Mode can handle some of these, but I haven't taken the time to get comfortable with Org-Mode's way of doing things. Instead, I use the more familiar calendar/diary functions in Emacs. Here are some basic preferences that aren't very interesting.

```
17a <Calendar 17a>≡
  (setq diary-file "~/diary"
        calendar-view-diary-initially-flag t
        calendar-mark-diary-entries-flag t
        diary-number-of-entries 7
        calendar-date-display-form '(year "-" month "-" day)
        calendar-time-display-form '(24-hours ":" minutes
                                     (if time-zone " (" time-zone (if time-zone ")"))
        diary-date-forms '((year "-" month "-" day)
                          (month "/" day "[^/0-9]")
                          (month "/" day "/" year "[^0-9]")
                          (monthname " *" day "[^,0-9]")
                          (monthname " *" day ", *" year "[^0-9]")
                          (dayname "\\W")))
        (add-hook 'diary-display-hook 'fancy-diary-display)
        (add-hook 'today-visible-calendar-hook 'calendar-mark-today)
```

Since I spend most of my time in Emacs, it helps to pop up reminders of appointments in advance. This brings them up in the mode line 12 minutes before the appointment (and every 3 minutes up until the appointment). I'd like to eventually have a better warning system for when I'm away from the computer, but for now this does the trick.

```
17b <Calendar 17a>+≡
  (appt-activate)
```

In case I need to jump to the calendar from somewhere else:

```
17c <Calendar 17a>+≡
  (global-set-key (kbd "C-c d") 'calendar)
```

Contacts Having my contacts stored in Emacs means that they're easily accessible in Gnus. I used to keep them in vCard format and may again at some point because BBDB lacks some features I'd like. As with Org-Mode, I like to use a more recent BBDB.

```
17d <Contacts 17d>≡
  (add-to-load-path "bbdb-2.35/lisp")
  (add-to-info-path "bbdb-2.35/texinfo")
  (require 'bbdb)
  (bbdb-initialize)
```

Uses [add-to-info-path 2c](#) and [add-to-load-path 2a](#).

By default, BBDB validates phone numbers. I store all my phone numbers with international dialing prefixes.

```
18a <Contacts 17d>+≡
      (setq bddb-north-american-phone-numbers-p nil)
```

Encryption Encrypting passwords and sensitive information is something I'll only trust with Free Software. I used to manage them with GPG on the command line. I then wrote an Emacs extension called jegpg. Finally, I came across EasyPG. It's what I was trying to accomplish with jegpg and more.

```
18b <Encryption 18b>≡
      (add-to-paths "epg-0.0.16")
      (require 'epa-setup)
      (setq epa-file-encrypt-to "chris@forno.us")
```

Uses `add-to-paths` 2e.

With EasyPG, I can visit a file (C-x C-f) with a .gpg extension and EasyPG will automatically encrypt it for me when I save it. Later, when I open it, EasyPG will decrypt it. The secret to making this usable and secure is to use `gpg-agent` and a pinentry program³.

Accounting I used to use `GnuCash` to keep track of my finances. While it's a good program and was doing its job, I wanted to try out `ledger`. The idea of working directly in a text file was appealing. So far, it's working pretty well. The Emacs extension for ledger adds some syntax highlighting and macros.

```
18c <Accounting 18c>≡
      (add-to-load-path "ledger")
      (require 'ledger)
```

Uses `add-to-load-path` 2a.

Sometimes I'll lose an unreconciled transaction somewhere in my ledger file which can be really confusing when reconciling on the command line. I wrote a function that allows me to go to the top of my ledger file and jump to each unreconciled transaction from there.

```
18d <Accounting 18c>+≡
      (defun ledger-search-forward-unreconciled ()
        "Search forward to the next unreconciled transaction.
        Literally searches for a date at the start of the line, not followed
        by an asterisk."
        (interactive)
        (search-forward-regexp "[0-9]\\{4\\}/[0-9]\\{2\\}/[0-9]\\{2\\} [*]")
        (goto-char (- (point) 1)))
```

Defines:

```
ledger-search-forward-unreconciled, never used.
```

³I use `pinentry-gtk-2`.

I bind it to C-c C-s since it reminds me of searching.

```
19a <Accounting 18c>+≡
      (add-hook 'ledger-mode-hook
        '(lambda () (local-set-key (kbd "C-c C-s") 'ledger-search-forward-unreconciled)))
```

Publishing I dislike word processors. I want to be assured that what I write will be editable decades from now. I also like to use revision control.

```
19b <Publishing 19b>≡
      (setq user-full-name "Chris Forno (jekor)")
      <Muse Mode 19c>
      <TeX 22b>
      <HTML Export 22c>
      <Graphviz 23a>
```

While T_EX is my preferred typesetting format, it can be a bit burdensome at times. All I really need most of the time is headlines, minor font changes, and lists. For those, Muse Mode is enough, and it allows me to get a realtime preview as I edit (which I think can be done for T_EX, but not as easily).

```
19c <Muse Mode 19c>≡
      (add-to-load-path "muse-3.12/lisp")
      (add-to-info-path "muse-3.12/texi")
      (require 'muse-mode)
      (require 'muse-html)
      (require 'muse-latex)
      (require 'muse-project)
      (require 'muse-book)
```

Uses add-to-info-path 2c and add-to-load-path 2a.

Currently I store everything in an `articles` directory. That'll change once I get more familiar with Muse Mode configuration (probably after I start actually publishing the articles). The articles go to my `public_html` directory which I then rsync to my server.

```
19d <Muse Mode 19c>+≡
      (setq muse-project-alist
        '(("articles"
          ("~/document/article" :default "index")
          (:base "html" :path "~/document/public_html")
          (:base "pdf" :path "~/document/public_html"))))
      <Footnotes 20a>
      <Muse UTF-8 20b>
      <Graphviz Tags 20c>
      <Muse LATEX Template 21a>
```

For some reason, I like footnotes. When I read a book, I always read the footnotes. I'm not sure why. Maybe nobody reads my footnotes, but at least it keeps my writing from having too many parenthetical statements.

Muse has a simple syntax for footnotes, but adding them while writing is easier with `footnote-mode`. Adding footnotes is as easy as `C-c ! a`.

```
20a <Footnotes 20a>≡
      (autoload 'footnote-mode "footnote" nil t)
      (add-hook 'muse-mode-hook 'footnote-mode)
```

I do all of my writing in the UTF-8 encoding. These lines tell Muse Mode to output the necessary meta information and to not try and escape non-ASCII (or non-iso-8859-1, I can't remember) characters.

```
20b <Muse UTF-8 20b>≡
      (setq muse-html-charset-default "utf-8"
            muse-html-encoding-default (quote utf-8))
```

Muse Mode has a neat feature that allows you to create arbitrary `<tags>` that you can process at publish time. I like to use `Graphviz` to make illustrations. Having a `<graphviz>` tags helps me keep all of my article in one place (I lose track of associated but separate files pretty easily).

The following function is called by `muse-publish` whenever it encounters a `<graphviz>` tag. It runs `dot2tex` on the tag body and converts it to `TikZ` format (a graphic language for `TeX`). Using `TikZ` means that the final graphics will be nice and scalable.

This of course assumes that the article will be published with `LATEX`. I don't currently have a solution for HTML.

```
20c <Graphviz Tags 20c>≡
      (defun muse-publish-graphviz-tag (beg end)
        (shell-command-on-region beg end "dot2tex --format tikz --figonly" nil t)
        (muse-publish-mark-read-only (region-beginning) (region-end)))

      (add-to-list 'muse-publish-markup-tags
                  '("graphviz" t nil nil muse-publish-graphviz-tag))
```

Defines:

```
muse-publish-graphviz-tag, never used.
```

For TikZ to actually work, the `tikz` package needs to be included in the \LaTeX header.

```
21a <Muse  $\LaTeX$  Template 21a>≡
    (setq muse-latex-header "\\documentclass{article}

    <My  $\LaTeX$  Packages 21b>

    \\usepackage{tikz}

    % These are needed for Graphviz shapes like ellipses.
    \\usetikzlibrary{decorations,arrows,shapes}

    <Custom Header 22a>
    ")
```

I've come to use a few \LaTeX packages regularly.

I sometimes use non-ASCII characters. UTF-8 seems to be the clear winner for Unicode, and fortunately \LaTeX supports it. It requires **both** `ucs` and `inputenc`.

```
21b <My  $\LaTeX$  Packages 21b>≡
    \\usepackage{ucs}
    \\usepackage[utf8]{inputenc}
    \\usepackage[english]{babel}
    \\usepackage[T1]{fontenc}
    \\usepackage{hyperref}
    \\usepackage[pdftex]{graphicx}
    \\usepackage[x11names,rgb]{xcolor}
```

By default, the `hyperref` packages outlines links with a colored box. I think it's distracting. Coloring the links should be enough.

```
21c <My  $\LaTeX$  Packages 21b>+≡
    \\hypersetup{colorlinks=true}
```

For reasons I don't understand, Muse Mode's doesn't do a good job of printing metadata about an article.

```
22a <Custom Header 22a>≡
  \begin{document}

  \title{<lisp>(muse-publishing-directive \"title\")</lisp>}
  \author{<lisp>(muse-publishing-directive \"author\")</lisp>}
  \date{<lisp>(muse-publishing-directive \"date\")</lisp>}

  \maketitle

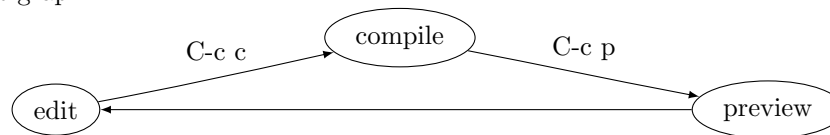
  <lisp>(and muse-publish-generate-contents
          (not muse-latex-permit-contents-tag)
          \"\\tableofcontents
  \\newpage\")</lisp>
```

Even though I do most of my writing in Muse Mode, I keep some documents (such as my resumé) in T_EX or L^AT_EX⁴ format. AUCT_EX comes in rather handy for syntax highlighting, even if I don't use its shortcuts.

```
22b <TEX 22b>≡
  (add-to-load-path "auctex-11.83")
  (add-to-info-path "auctex-11.83/doc")
  (load "auctex.el" nil t t)
```

Uses `add-to-info-path` 2c and `add-to-load-path` 2a.

Graphviz I'm a big fan of **Graphviz**. Even though it's limited to graphs and diagrams, it's amazing how frequently it's useful for expressing ideas. Of course, the only problem is that it takes me a while to figure out how to get a graph to look good. Luckily there's an Emacs mode that allows me to quickly preview the graph.



HTML Export is supported by Org-Mode via the `htmlize` package. It takes an Emacs buffer, figures out the font decorations, and converts it to HTML.

```
22c <HTML Export 22c>≡
  (require 'htmlize)
```

⁴I'm currently looking into alternatives like ConT_EXt.

```
23a <Graphviz 23a>≡
      (add-to-load-path "graphviz-dot-mode")
      (load "graphviz-dot-mode.el")
```

Uses `add-to-load-path` 2a.

Graphviz mode seems to have a broken automatic indentation implementation. This doesn't fix it, but it makes it better.

```
23b <Graphviz 23a>+≡
      (setq graphviz-dot-indent-width 2)
      (setq graphviz-dot-auto-indent-on-semi nil)
      (setq graphviz-dot-auto-indent-on-braces nil)
```

Mathematics For most simple calculations, Emacs's Calc (M-x `calc`) suffices. But sometimes I need more.

```
23c <Mathematics 23c>≡
      <Maxima 23d>
      <GNU R 23f>
```

Maxima `Maxima` can be used on the command line, but it doesn't look nearly as good as in `imaxima`. Basically, `imaxima` converts Maxima's output into \LaTeX -formatted graphics. Make sure that your monitor dimensions are set before you try, or you'll get distorted output.

```
23d <Maxima 23d>≡
      (add-to-paths "imaxima-imath-1.0b2")
      (autoload 'maxima "maxima" "Frontend for maxima" t)
      (autoload 'imaxima "imaxima" "Image support for Maxima." t)
```

Uses `add-to-paths` 2e.

The default font size is too small for my liking (as with most default font sizes).

```
23e <Maxima 23d>+≡
      (setq imaxima-fnt-size "huge")
```

GNU R While Maxima is great for symbolic work, `R` is better for statistics. “Emacs Speaks Statistics” (ESS) makes working with R a little bit more convenient.

```
23f <GNU R 23f>≡
      (add-to-load-path "ess-5.3.3/lisp")
      (require 'ess-site)
```

Uses `add-to-load-path` 2a.

Programming The reason I started using Emacs was for programming. Strangely, nowadays I spend the majority of my time in Emacs working on other tasks. But programming is still important and requires a lot of configuration to fit my style.

I generally work with these languages:

Haskell is my language of choice. I use it for most heavy lifting when a more domain-specific language doesn't exist.

PHP I use in my day job.

JavaScript for when occasionally do client-side web programming.

Lisp is neat, but nowadays I only write Elisp for Emacs.

I also work with the following non-programming languages, which I include in the programming section since the act of editing them is similar.

CSS for which hex color mode was made.

SQL I treat specially. To me it is more than just a language to be used in strings in a host language.

```
24a  <Programming 24a>≡
      <Indentation 24b>
      <Haskell 26a>
      <PHP 27a>
      <SQL 27c>
      <Commenting 27b>
      <Diffing 27e>
```

Indentation I finally chose the side of spaces in the tabs-vs-spaces war. What helped me make the decision was working with languages that have non-trivial indentation (i.e. non-ALGOL-derived languages). It would be ideal to have some sort of elastic tabs to work with, but for now it's fixed-width fonts and spaces.

```
24b  <Indentation 24b>≡
      <Convert Tabs to Spaces 24c>
      <Set Tab Width 25a>
      <Disable Automatic Indentation 25b>
      <Filling 25c>
```

Here we setup tabs-to-spaces for all modes that use indentation.

```
24c  <Convert Tabs to Spaces 24c>≡
      (setq-default indent-tabs-mode nil)
```

I prefer 2 spaces for indentation. This is only necessary for languages with ALGOL-style syntax.

```
25a <Set Tab Width 25a>≡
      (setq-default tab-width 2)
      (setq-default c-basic-offset 2)
```

Some modes try to be too intelligent with automatic indentation. Unfortunately, it doesn't work too well for me. I spend more time correcting indentation than I would be just indenting each line manually, so I turn it off.

```
25b <Disable Automatic Indentation 25b>≡
      (add-to-hooks '(lambda () (setq c-syntactic-indentation nil))
                   'php-mode-hook
                   'javascript-mode-hook)
```

Uses `add-to-hooks` 5a.

Filling While not exactly indentation or limited to programming, the fill width—the (soft) limit on the number of characters in a line—fits in nicely here. I stick to the traditional 79-character limit. While I don't expect to have to read my programs or other text files on an 80-character terminal except in very rare situations, I do think that too much text on a single line becomes more difficult to read.

Additionally, even with a 20" monitor and a fully maximized Emacs, I use a font large enough to allow less than 100 characters on a line anyway.

This does require that I break my programming lines down when I'd rather not, but I've found that with good alignment and indentation the code usually looks better. Additionally, it has forced me to think about ways of simplifying my code: reducing nested indentation, breaking down long chains of function calls, etc.

```
25c <Filling 25c>≡
      (setq-default fill-column 79)
```

Autofilling and the `fill-paragraph` command (`M-q`) both will avoid breaking a line after a period if it's not considered the end of a sentence. Well, I long ago got out of the habit of typing 2 spaces at the end of a sentence—even when using monospaced fonts.

```
25d <Filling 25c>+≡
      (setq sentence-end-double-space nil)
```

Haskell For writing Haskell code, **Haskell-mode** is a must. It has some really awesome indentation assistance, neat echo-area documentation, and the ability to interact with a Haskell interpreter.

```
26a <Haskell 26a>≡
      (add-to-paths "haskell-mode-2.4")
      (load "haskell-site-file")
      (add-hook 'haskell-mode-hook 'turn-on-haskell-doc-mode)
      <Haskell Indentation 26b>
```

Uses `add-to-paths` 2e.

Haskell indentation is pretty complex. This could use some enhancement.

```
26b <Haskell Indentation 26b>≡
      (setq haskell-indent-offset 2)
      (add-hook 'haskell-mode-hook 'turn-on-haskell-indent)
```

I want Haskell-mode activated for Haskell source (`.hs`), interface (`.hi`), and literate⁵ source (`.lhs`) files.

```
26c <Haskell 26a>+≡
      (setq auto-mode-alist
            (append auto-mode-alist
                    '(("\\.hs$" . haskell-mode)
                      ("\\.hi$" . haskell-mode)
                      ("\\.lhs$" . literate-haskell-mode))))
```

Haskell can be interpreted or compiled. Hooking into a Haskell interpreter can come in handy for interactive development. Haskell-mode supports a basic comint-style integration with an interpreter. I personally prefer GHCi with GHC extensions enabled.

```
26d <Haskell 26a>+≡
      (setq haskell-program-name "ghci -fglasgow-exts --interactive")
```

Haskell mode can search **Hoogle** for function definitions. I suspect there's a more general way of doing this that can be applied to other modes as well, but I haven't taken the time to figure out how yet. Since this is a new feature, it has no default key binding.

```
26e <Haskell 26a>+≡
      (add-hook 'haskell-mode '(lambda () (local-set-key (kbd "C-c g"))))
```

⁵Haskell has built-in literate programming support. While it may not fit the definition of literate programming exactly—it still requires programs to be written in a certain order—it has a nice pretty printer and works passably. Until there is a better tool, I'll continue to use it.

PHP We need to tell PHP what comment syntax to use. I use nothing but `//`. Using `/* */` isn't necessary if you have good region-based comment toggling set up.

```
27a <PHP 27a>≡
      (add-hook 'php-mode-hook
                '(lambda () (setq comment-start "// "
                                   comment-end "")))
```

While commenting is not limited to programming, it's while programming that I use it most. All I do is bind the `comment-region` command to `C-c c`. This gives me `uncomment-region` as `C-u C-c c` for free.

```
27b <Commenting 27b>≡
      (global-set-key (kbd "C-c c") 'comment-region)
```

SQL I've tried to use the various graphical database administration tools. While they have some nice features, they're usually broken in a number of annoying ways. I'd rather work directly in the terminal-based client program for the database. SQL mode allows me to do that and get some of the conveniences of Emacs editing. I can also send SQL statements from a buffer to the database.

```
27c <SQL 27c>≡
      (require 'sql)
```

Sometimes I have an ongoing session with a database while designing a schema. It's useful to keep a history of the commands I issued in case I need to review them later.

```
27d <SQL 27c>+≡
      (setq sql-input-ring-file-name (concat *emacs-path* "/sql.history"))
```

Uses `*emacs-path*` [1b](#).

Diffing is something I don't have to do often (at least not in the case of having 2 files to try and reconcile), but when I do I'm very glad for Ediff. I always run Emacs in a single frame, so I don't want Ediff to mess with that.

```
27e <Diffing 27e>≡
      (setq ediff-window-setup-function 'ediff-setup-windows-plain)
```

Automation is closely related to programming and I use it mostly for programming. But it is a separate concept.

28a `<Automation 28a>≡`
`<Pair Completion 28b>`
`<Templates 28d>`
`<Date Updating 28e>`

The most prevalent automatic assistance I get from Emacs is completing the pair of a logic group for me. Many modes already support what're called skeleton pairs.

28b `<Pair Completion 28b>≡`
`(setq skeleton-pair t)`

Some pairs are useful for me everywhere, so instead of relying on individual modes, I bind them globally.

28c `<Pair Completion 28b>+≡`
`(global-set-key (kbd "(") 'skeleton-pair-insert-maybe)`
`(global-set-key (kbd "[") 'skeleton-pair-insert-maybe)`
`(global-set-key (kbd "{") 'skeleton-pair-insert-maybe)`
`(global-set-key (kbd "\"") 'skeleton-pair-insert-maybe)`

The next form of automation is templates. I store all of my templates in `~/.emacs.d/auto/`.

28d `<Templates 28d>≡`
`(auto-insert-mode 1)`
`(setq auto-insert-directory (concat *emacs-path* "/auto/"))`

Uses `*emacs-path*` 1b.

Finally, the one I couldn't live without: automatic timestamp and copyright updating on save. I would never remember to update the dates without these—and not all my files are under revision control.

I use a timestamp format similar to ISO8601 which includes my name and handle (`user-full-name`).

28e `<Date Updating 28e>≡`
`(add-hook 'before-save-hook 'time-stamp)`
`(add-hook 'before-save-hook 'copyright-update)`
`(setq time-stamp-format "%:y-%02m-%02d %02H:%02M:%02S %Z %U")`

Internet It's true: you can do almost anything within Emacs. I draw the line at web browsing⁶ though.

29a *<Internet 29a>*≡
<Email and News 29b>
<IRC 31c>
<Web 31d>

Email and News are text at their best. I don't need images or fancy fonts in either. And I like to write my replies in Emacs.

29b *<Email and News 29b>*≡
<Gnus 29c>
<Sending Mail 31a>

Gnus So I heard about this **Gnus** program and decided to give it a try. For a reason I can't remember, I was tired of Thunderbird and I was probably just curious. It does have serious limitations in the way it treats email as just another news source. But for now, it works.

29c *<Gnus 29c>*≡
<Email 29d>
<Mailing Lists 30c>
<Gnus Preferences 30d>
<MIME 30e>

Email I don't check my email more than once a day, and for that sort of usage, Gnus is usable. Out of laziness, my IMAP server is not my primary source of "news". I configured it after using Gnus for news/Usenet for a while, so it's one of my secondary "select methods".

29d *<Email 29d>*≡

```
(setq gnus-secondary-select-methods
      '( (nnimap "jekor.com"
                (nnimap-address "mail.jekor.com")
                (nnimap-stream tls)
                (nnimap-list-pattern (<Email Inboxes 29e>))
                (<Usenet 30b>)))
      (<Sent Email 30a>))
```

Using Gnus with an IMAP server is a little tricky. You need to know the server's folder naming convention and where your new mail comes in.

29e *<Email Inboxes 29e>*≡
 ("INBOX" "mail/*" "INBOX.*")

⁶Emacs does have a couple web browsers, but they are both unmaintained and broken.

Handling sent email is also a bit awkward. Gnus treats it as an “archived” message.

```
30a <Sent Email 30a>≡
      (setq gnus-message-archive-method '(nnimap "jekor.com")
            gnus-message-archive-group "INBOX/Sent")
```

Occasionally when I’m bored I’ll read Usenet, which doesn’t require much configuration.

```
30b <Usenet 30b>≡
      (nntp "nntp.aioe.org")
```

But I primarily read some mailing lists archived on gmane.org, as the Usenet hierarchy seems to have calcified.

```
30c <Mailing Lists 30c>≡
      (setq gnus-select-method '(nntp "news.gmane.org"))
```

The Gnus preferences are a bit boring, and you can look them up for yourself with `C-h v`.

```
30d <Gnus Preferences 30d>≡
      (setq gnus-check-new-newsgroups nil
            gnus-save-newsrc-file nil
            gnus-read-newsrc-file nil
            gnus-save-killed-list nil
            gnus-group-sort-function 'gnus-group-sort-by-level
            gnus-dribble-directory *emacs-path*
            gnus-default-charset 'utf-8)
```

Uses `*emacs-path*` [1b](#).

Handling attachments with Gnus hasn’t gotten much easier than when you had to know what MIME was in order to work with them. Back then, you had a `~/mailcap` file that told MIME-aware programs how to handle attachments. Well so do I.

```
30e <MIME 30e>≡
      (add-hook 'gnus-summary-mode-hook 'mailcap-parse-mailcaps)
      (setq mailcap-download-directory "~/sandbox")
```

I've noticed that a lot of ISPs have taken to blocking outgoing traffic on port 25. OK. The first line here just tells Emacs that I want to use the SMTP protocol to send mail.

```
31a  <Sending Mail 31a>≡
      (setq message-send-mail-function 'smtpmail-send-it
            smtpmail-smtp-service 2525
            smtpmail-default-smtp-server "mail.jekor.com"
            smtpmail-auth-credentials '("mail.jekor.com" 2525 "chris" nil))
            smtpmail-starttls-credentials '("mail.jekor.com" "2525" nil nil))
```

I like Gnus's email composer, and I like to be able to access it from anywhere.

```
31b  <Sending Mail 31a>+≡
      (global-set-key (kbd "C-c m") 'gnus-group-mail)
```

IRC There's not much to say about IRC with Emacs. ERC (M-x `erc`) works well with its default settings. I only need to tell it my nick⁷. Maybe I'll get around to auto-nickserver-identify at some point.

```
31c  <IRC 31c>≡
      (setq erc-nick "jekor")
```

Web While I don't use Emacs to browse the web, I do come across URLs in buffers I'm working on. Many Emacs extensions have support for the idea of sending the URL to another program. I like to open URLs in new tabs in **Firefox**.

```
31d  <Web 31d>≡
      (setq browse-url-browser-function 'browse-url-firefox
            browse-url-new-window-flag t
            browse-url-firefox-new-window-is-tab t)
```

Music I've tried dozens of audio players for GNU/Linux. Most of them don't support all of the formats I use (**module files**, **Vorbis**, **FLAC**, and **MP3**). **MPD** not only supports them all, but has a nice client/server architecture.

Mingus is a simple no-frills interface to MPD. It requires libmpdee.

```
31e  <Music 31e>≡
      (add-to-load-path "mingus-0.27")
      (require 'libmpdee)
      (require 'mingus)
      <Mingus Key Bindings 32a>
```

Uses `add-to-load-path` [2a](#).

⁷You can sometimes find me on irc.freenode.net

I don't want to always have to switch to the Mingus buffer to do a couple common operations. I want to be able to pause and play and to move to the previous and next song. To do so, I need some global key bindings.

```
32a <Mingus Key Bindings 32a>≡
      (global-set-key (kbd "C-x p") 'mingus-toggle)
      (global-set-key (kbd "C-x <") 'mingus-prev)
      (global-set-key (kbd "C-x >") 'mingus-next)
```

Pictures OK, so browsing pictures is not something I imagined doing with Emacs. I tried it out for fun, but it turns out to be quite useful sometimes. While I still use **GQview** most of the time, I did configure the layout of **Image-Dired**.

```
32b <Pictures 32b>≡
      (setq image-dired-thumb-height 200
            image-dired-thumb-size 200
            image-dired-thumb-width 200
            image-dired-thumbs-per-row 10)
```

If you'd like to convert this document into a usable .emacs file, download the source and Makefile from <http://jekor.com/emacs/dot-emacs.tar.gz>.