

Emacs Guide

Taylor Venable

December 7, 2009

Abstract

How to do a lot of things in Emacs.

Contents

Typographical Conventions	3
1 Introduction	3
1.1 Sources of Help	3
I Emacs Guide	3
2 Getting To Know Emacs	3
2.1 Swap Those Control & Caps Lock Keys!	4
2.1.1 Using xmodmap	4
2.1.2 Using setxkbmap	4
2.1.3 Swapping Control And Caps Lock On The BSD Virtual Terminal	4
3 Unicode & XFT Support	5
4 Fontifying Special Words	5
5 Handling A Bunch Of Major Modes	6
6 Using Exuberant Ctags for Speedbar	7
7 Mail, USENET, and IRC	7
7.1 Reading Mail Messages	7
7.2 Sending Mail Messages	8
7.3 Mail Aliases	9
7.4 NNTP Troubles	9
8 The Emacs Multimedia System	10
8.1 Installation	10
8.2 Keybindings	10
8.3 Players	10
8.4 Modeline Display	11
8.4.1 Track Description	11
8.5 Adding Tracks from Dired	11
8.6 Metadata Loading	12
II Implementation	12

9 Pre-Load Setup	12
9.1 Load Path	13
9.2 Conditional Load Variables	13
9.3 Info Directories	13
10 Major Mode Setup	13
11 Other Packages to Load	14
12 Personal Variables	16
12.1 X11 Fonts	16
12.2 TrueType Fonts	17
13 Utility Functions	20
13.1 Hex Decoding	20
13.2 Longest Line	20
13.3 Protected Require	21
14 File Management	21
14.1 Boilerplate & Templates	21
14.2 Executable Scripts	22
14.3 Coding Conventions	22
14.4 Path Functions	22
14.5 Find Changes File	23
14.6 Add Changes Entry	23
15 Major Mode Customization	24
15.1 All Modes	24
15.2 C Mode	25
15.3 Java Mode	25
15.4 Haskell Mode	25
15.5 HTML Mode	25
15.6 L ^A T _E X Mode	26
15.7 Python Mode	26
15.8 Ruby Mode	26
15.9 Org Mode	26
16 Server	26
A Colophon	27

Typographical Conventions

- Standards are presented in a sans-serif face.

1 Introduction

Emacs (short for Editing Macros, or sometimes alternatively, Escape Meta Alt Control Shift) are probably the most powerful editors ever created by humankind. It is the Evangelion of programmers' programmable editors. You can use it to write code, read email, and perform psycho-analysis. Emacs can even save your soul if you choose to embrace it. So why exactly is Emacs so great?

“Calling EMACS an editor is like calling Earth a hunk of dirt.”

— Chris DiBona

Well, it's mostly because Emacs is written largely in its own dialect of Lisp, and as such has at its core a specialized Lisp interpreter. This causes it to weigh in a bit heavily (the CVS version with full debugging can run at over 48 MB of RAM after some extended use) but also gives the benefit of being extensible to an unprecedented level. Some view this as being a bad thing, and happily stick with slimmer editors like vi. I myself am perfectly alright with memory usage in the 32 to 64 meg range, because frankly I think it's more than worth it. Now, I have used both Emacs and Vim for several years, and I still frequently use Vim for quick editing. But when I really get down to development brass tacks, it's GNU Emacs that I choose.

The Emacs configuration file, read at startup to restore user customizations, is interpreted in Emacs Lisp. As a result, writing complex configurations requires some knowledge of how Emacs Lisp works. If you've ever used a functional programming language before, like Scheme or Common Lisp, you'll find that Emacs Lisp is quite similar in terms of language architecture. The only big differences are the wide variety of specialized functions available in Emacs Lisp. As I've been working hard on trying to master Emacs, I've got a nice little configuration file built up from my experiments. Go ahead and use it to follow along with these examples, or as a basis for your own Emacs customization files. (Please note that some of the examples below have been refactored and in some cases combined in my configuration file.) Sites About Emacs

For those new to Emacs, here are a few websites that describe the editor family (and specific implementations) in more detail:

* Wikipedia Page * GNU Emacs Project * EmacsWiki

1.1 Sources of Help

For help using Emacs, there are many options. One is to use the mailing list for whatever version you're using. Those with GNU Emacs can use help-gnu-emacs@gnu.org. Those interested in posting bugs or reading more internal news, subscribe to emacs-devel@gnu.org. If you're using an Emacs project, such as the GNUS mail and news reader, try subscribing to that project's mailing list: ding@gnus.org.

There is also an IRC channel on FreeNode: #emacs. Numerous other Emacs projects have IRC channels as well, for example #gnus (also on FreeNode).

Part I

Emacs Guide

2 Getting To Know Emacs

The greatness of Emacs can, to the uninitiated, be astonishing to the point of paralysis. Where does one get started? Well, first you've got to learn the keystrokes. When you see "C-x" that means hold the control key and press x. The upper-case C here is called an accelerator. Another popular accelerator is "M" for meta

(that is, Alt for most people or Option for Apple users). If you want to, you can prefix the next keystroke by pressing escape, rather than holding down meta. In the event of a longer keystroke sequence, like "C-x C-f" just do the first stroke then perform the second stroke. Simple, right?

To really start learning Emacs, try hitting "C-h t" for the Emacs learn-by-doing tutorial. Once you've worked your way through that, you're pretty much good to start editing. Because Emacs is entirely self-documenting, help can be obtained at any time for nearly any function, variable, or keybinding. For example, to find out what keys currently do what, type "C-h b" (all the help keybindings start with "C-h"). And then that's pretty much it. You can happily type your day away and whenever you get stuck, just invoke one of these help keystrokes to find out what your mind is lacking.

2.1 Swap Those Control & Caps Lock Keys!

After a few days of otherwise joyful Emacs use, you may find that your pinky finger screams in anguish every time you reach for the control key. Since Emacs makes frequent use of control sequences, avoiding pain and the danger of repetitive injury strain (an ailment that afflicts some of the greatest Unix hackers) is very important. Nearly all real Unix keyboards (like the incredibly cool Happy Hacking Keyboard) put the all-important control key on the home row to help alleviate these problems, but PC keyboards annoyingly stick the control key in the very farthest corners, making life worse for Emacs users. But if you're on Unix, fixing this problem is easy: just remap the keys!

2.1.1 Using `xmodmap`

If you're using the X window system, you can use the `xmodmap` program to remap your keyboard. Doing this is fairly straight forward, just place the following lines into your `~/.xmodmaprc` file:

```
! swap left control and caps lock

remove Lock = Caps_Lock
remove Control = Control_L

keysym Control_L = Caps_Lock
keysym Caps_Lock = Control_L

add Lock = Caps_Lock
add Control = Control_L
```

Now add the following line to your `.xinitrc` or `.xsession` file: "`xmodmap ~/.xmodmaprc`". The next time you start X, pressing your left control key will set the caps lock, and holding the caps lock key will set the control accelerator. Yippie!

2.1.2 Using `setxkbmap`

The program `setxkbmap` can do a lot of cool things, including swapping the Control and Caps Lock keys. To do so, use the following incantation:

```
setxkbmap -option ctrl:swapcaps
```

2.1.3 Swapping Control And Caps Lock On The BSD Virtual Terminal

To switch the functions of your caps lock and control keys on the BSD virtual terminal, you need to play with the keymap used by the console system in use. On systems like NetBSD and OpenBSD using `wscnsc` can very easily do this through the `wscnscctl` program. English keyboard layouts have builtin options for swapping the caps lock and control keys. These are specified with the extension `".swapctrlcaps"` to the normal keymap. To use them, set the keyboard encoding value in NetBSD like so: "`wscnscctl -w encoding=us.swapctrlcaps`" or in OpenBSD like this: "`wscnscctl keyboard.encoding=us.swapctrlcaps`".

If on the other hand, you happen to be using syscons on FreeBSD, the process is a little more involved, but still very easy. Just find the file that corresponds to the keymap you are currently using in the `/usr/share/syscons/keymaps` directory. Copy this file and swap the lines that indicate the left control and caps lock keys. If you're using the US ISO keymap, left control is keycode 029 and caps lock is keycode 058. Swap these code values so that caps lock is set to code 029 and left control is code 058. Then save the result into a new file in the same directory, maybe with a name like `"us.swapctrlcaps.kbd"`. Now use `kbdcontrol` to install the new keymap: `"kbdcontrol -l us.swapctrlcaps"`. If you're shy about editing important files (what?) then you can download my swapped keymap based on the US ISO keymap.

3 Unicode & XFT Support

The very latest development version of Emacs (this has a version number starting with 23) has continued new Unicode support, as well as experimental support for XFT fonts. To check out these developments and see how they work for your system, first check out the code from CVS, using the `emacs-unicode-2` branch.

```
cvs -z3 -d:pserver:anonymous@cvs.savannah.gnu.org:/sources/emacs co -r emacs-unicode-2 emacs
```

Now run `./configure` like usual, but make sure to add the options that enable XFT fonts. To do this, you must be building with GTK support. Then add the options `--enable-font-backend` and `--with-xft`. Lastly, build like usual (using `make bootstrap` and `make install`).

You also need to enable XFT support at runtime by passing the `--enable-font-backend` support to Emacs when you start it. (Note: In 23.0.50 and newer, you no longer have to supply this option at run time.) You'll also have to set the font, either by passing it on the command line (via the `--font` option) or through your X resources file. To use the latter method, put the following into your `.Xresources` file and run `xrdb` like usual:

```
Emacs.font: DejaVu Sans Mono-10
Now sit back and enjoy those beautiful smoothed fonts!
```

4 Fontifying Special Words

Vim and some other editors have builtin support for highlighting special words like `TODO` and `FIXME` in source code files. It's somewhat surprising, considering everything can do right out of the box, that Emacs doesn't have this feature by default. But like pretty much anything else in Emacs you can add it yourself with little hassle.

Essentially, what you have to do is add a special font-lock keyword for the phrases you want to highlight. This is easily enough done with the `font-lock-add-keywords` function, but we run into a little problem when using this method. It can only change one mode at a time! Now it's Lisp to the rescue with the `mapcar` function, which takes a given function and applies it to each element in a given list. Remembering to set up a separate face for our special words first, we can do something like this:

```
(make-face 'special-words)
(set-face-attribute 'special-words nil
  :foreground "White" :background "Firebrick")
(let ((prog-modes '( c-mode c++-mode java-mode ada-mode sh-mode tcl-mode
  cperl-mode python-mode ruby-mode lisp-mode ))
  (pattern "\\<\\(FIXME\\|TODO\\|NOTE\\|WARNING\\):"))
(mapcar
  (lambda (mode)
    (font-lock-add-keywords mode '((,pattern 1 'special-words prepend))))
  prog-modes))
```

Notice the list `prog-modes` which contains all the modes we want to highlight these words for. The regex we're using is stored in the variable `pattern`. Then it's a simple matter of running each of these modes in turn through the lambda function we provide to activate highlighting on the specified pattern.

5 Handling A Bunch Of Major Modes

Since the main power of Emacs lies in its extensibility, you may find (over time) that you build up quite a collection of things to autoload for different file extensions. For example, I have external major modes for CSS, FVWM, Lua, Perl, PHP, Prolog, Python, and Ruby. It's a complete pain to have to enter these all manually into Emacs' configuration, and luckily with a little work you can streamline it easily.

```
(mapcar
(lambda (mode-list)
(progn
(autoload (car mode-list) (symbol-name (car mode-list)))
(concat "Major mode for editing " (cadr mode-list) "source code."))
(add-to-list 'auto-mode-alist
'((,(cadr (cdr mode-list)) . ,(car mode-list))))))
'((css-mode      "Cascading Style Sheets"  "\\\.css$")
(fvwm-mode      "FVWM Configuration"      "\\\.fvwm/config$")
(lua-mode       "Lua"                      "\\\.lua$")
(cperl-mode     "Perl"                     "\\\.p[lm]$" )
/php-mode       "PHP"                      "\\\.php$")
/prolog         "Prolog"                    "\\\.pdb$")
/python-mode    "Python"                   "\\\.py$")
/ruby-mode      "Ruby"                      "\\\.rb$")))
```

This is similar to the snippet above for fontifying special words in source code. Here we apply a lambda function to each element in a list of lists. Each sublist (as it were) contains the symbol for the mode, a very brief string description, and a regexp to match for autoloading it. Simple enough, right? Automatic Boilerplates

Emacs provides support for an auto-insert-mode to automatically add boilerplate code to a new file. Which file is used depends on the auto-insert-alist variable, which is a list of associated lists. There are two ways to do the dotted-list, which you can read about in the documentation for the variable; I use the (REGEX . COMMAND) form. Here we go!

```
(require 'autoinsert)
(setq auto-insert-directory "~/templates/")
(setq auto-insert-query nil)
(defun template-list (insert-directory)
(let (lst)
(dolist (elt (directory-files insert-directory nil "[^\\.]" nil) lst)
(setq lst (append lst '((concat "\\." elt "$") . ,elt))))))
(setq auto-insert-alist (template-list auto-insert-directory))
(auto-insert-mode))
```

The first line requires that auto-insert support be present. The second sets the directory where all the boilerplate files are stored. The third line indicates that boilerplates should be automatically inserted without prompting.

The template-list function is really the heart of the customization. It takes the boilerplate directory as it's parameter, and creates a null list to hold what will become auto-insert-alist. For each file in the directory that doesn't start with a dot, it adds a dotted-list to the auto-insert-alist: this dotted-list consists of regex matching the filename extension, and the associated file to use as a boilerplate. Since the templates are named in files which match the extension, only a bit of manipulation is necessary to come up with a regex via the concat function. Automatic Indentation

If you write a lot of code in Emacs, you'll probably tire quickly of having to hit tab after starting each new line in your source. To get around this, you could teach yourself to use "C-j" instead of RET to insert newlines, but that's somewhat counter-intuitive. Isn't there a way to just auto-indent code? Yes, there is. Just rebind the return key to whatever "C-j" normally does. The lookup-key function will help you figure out what function to bind to.

One might ask why we don't simply swap the values for "C-j" and RET. The answer is that if we do that, nested hook calls (like c-mode-hook before java-mode-hook) will cause the swap to happen twice, the second reversing the effects of the first. If you want, you can assign the value of RET to an unbound key like "C-' (that is, control apostrophe) before reassigning RET, or just simply use "M-x newline" whenever you want to insert only a newline.

Using local-set-key we can reassign the RET key to whatever is normally invoked by pressing "C-j". To figure out exactly what that is, we check two keymaps: the mode-specific one and the global one. Now, I've seen some people just substitute newline-and-indent for the lookup-key function here (which is the global keymap binding), but some modes (like Ruby mode) bind their own special function for "C-j". Since that's the effect we're going for here, that's the method I've used. Hence, we check the local mode keymap before the global one.

```
(mapcar
(lambda (mode)
(let ((mode-hook (intern (concat (symbol-name mode) "-hook"))))
(mode-map (intern (concat (symbol-name mode) "-map"))))
(add-hook mode-hook
'(lambda nil
(local-set-key (kbd "RET")
(or (lookup-key ,mode-map "\C-j")
(lookup-key global-map "\C-j"))))))))
'(ada-mode c-mode c++-mode cperl-mode emacs-lisp-mode java-mode html-mode
lisp-mode php-mode ruby-mode sh-mode sgml-mode))
```

Unfortunately, this method creates a little more work for us, because we have to keep track not only of the mode to use, but also the keymap we're looking at. To keep from having to explicitly type both of these in the list parameter to mapcar, I've done a bit of symbol magic. (This took me almost two hours to finally figure out so I'm pretty proud of it.) Inside the mapcar function we create two variables mode-hook and mode-map which evaluate to symbols that represent the appropriate hook and keymap, respectively. In constructing these symbols, we take the current value from mapcar and create a symbol by appending either "-hook" or "-map" to the end of it. We can safely use this to determine the correct hook and keymap symbols for a given mode because of the convention that nearly everybody uses for defining mode hooks and keymaps. For example, when mapcar gives us lisp-mode we convert that to a string, stick "-map" onto the end of it, intern a new symbol from the result, and assign that symbol to the mode-map variable. Then we evaluate this variable in the body of the lambda expression (that will be added to the hook) and hence make the correct named keymap (lisp-mode-map) appear verbatim inside the hook's lambda!

6 Using Exuberant Ctags for Speedbar

Alright, we can all agree that Etags is great, but quite frankly, Exuberant Ctags is better. But of course, Emacs is so configurable that you can have Speedbar use whichever tagging facility your heart desires. Setting it up to use Exuberant Ctags instead of the default Etags is simple enough:

```
(setq speedbar-use-imenu-flag nil)
(setq speedbar-fetch-etags-command "/usr/bin/ctags-exuberant")
(setq speedbar-fetch-etags-arguments '("-e" "-f" "-"))
```

Set these parameters before you launch Speedbar.

7 Mail, USENET, and IRC

7.1 Reading Mail Messages

There are several different ways to read your email from Emacs, the most popular of which is GNUS. But GNUS is by design a news reader, meaning that there are some hurdles to overcome if you are familiar

with more standard email readers like Mutt or Thunderbird. The first thing you have to do is tell GNUS how to fetch and store your email; this is done by setting the `gnus-select-method` variable. (Actually this variable represents the primary method for getting messages; secondary methods are set through the `gnus-secondary-select-methods` variable.) Next you can set where your mail goes via the `nnmail-split-methods` variable. This is a list whose elements are lists which contain a target group and a regex to match. Any messages matching the regex will be stored in the given group. The last entry should have an empty regex so all remaining mail will go to that group.

Since GNUS behaves like a news reader, you have to subscribe to the groups which will receive email. You can subscribe to a group by using `S s` and then typing the name of that group. Note that by default only groups with unread messages will be displayed in the group buffer. To show all groups (both read and unread) press `L`. You can select a particular group with `RET` and unsubscribe from it by placing the cursor on its line and pressing `u`.

```
(require 'gnus)

(setq gnus-directory "~/email/")
(setq gnus-fetch-old-headers t)
(setq gnus-inhibit-startup-message t)
(setq gnus-outgoing-message-group "sent")
(setq gnus-select-method '(nnfolder ""))
(setq gnus-treat-display-smileys nil)

(setq nndraft-directory "~/email/")

(setq nnmail-split-methods
  '(("spam" "^Subject: \\[SPAM\\]")
    ("inbox" "")))

(setq gnus-group-line-format "%M%S%pP%3N / %3R : %(%-16G%)\\n")

(add-hook 'gnus-group-mode-hook
  #'(lambda nil (setq show-trailing-whitespace nil)))
(add-hook 'gnus-summary-mode-hook
  #'(lambda nil (setq show-trailing-whitespace nil)))
(add-hook 'gnus-article-mode-hook
  #'(lambda nil (setq show-trailing-whitespace nil)))
```

Here's an example configuration. I put all of these definitions into my `.emacs` file; some of these variables (like `inhibit-startup-message`) have to be defined there, but others don't. The hooks at the end are to shut off trailing whitespace display when using GNUS, because for some reason a lot of email has whitespace at the end of lines.

7.2 Sending Mail Messages

Compared to reading mail, sending it is rather easy to set up. First, you need to determine how you'll be sending your message: using a local program (like `sendmail`) or through SMTP (to a remote host). If you answered the former, you're pretty much all set right out of the box! Check the `customize / applications / mail buffer` for configuration. But if you chose the latter, transmitting mail via an external SMTP host, you've got a bit more work to do.

If your SMTP host requires authentication (which most seem to do nowadays), and if you're using GNU Emacs 21 or older, you need to fetch some more up-to-date Elisp files. Simon Josefsson and Stephen Crane-field have made modifications to the Emacs `smtpmail` source to add support for SMTP authentication. You can find information on <http://josefsson.org/emacs-rfc2554.html> Josefsson's Emacs RFC-2554 page. It basically boils down to downloading and enabling three different Emacs

Lisp source files. Just grab them from Emacs CVS and place them into the same directory; then prepend their location to the beginning of your load-path. When their functionality is required, the new files will be used instead of the old ones. (To boost speed a bit so its more on par with the builtin smtpmail functionality, you can call byte-compile-file on them from Emacs.) Now you can use SMTP authentication to send email messages!

Sending email via SMTP requires some configuration in your `.emacs` file. Here's a decent example:

```
(setq message-send-mail-function 'smtpmail-send-it)
(setq smtpmail-default-smtp-server "smtp.host.name")
(setq smtpmail-local-domain "local.host.name")
;; (setq smtpmail-debug-info t) ; puts debug info in *trace ...* buffer
(setq smtpmail-auth-credentials '(("smtp.host.name" 25 "user" "password")))

(setq mail-host-address "local.host.name")
```

This is pretty self-explanatory. The debugging functionality is helpful if you run into some kind of an error from the SMTP server, as the `*trace ...*` buffer prints out the entire transcript from the client/server interaction. You can now use the command `message-mail` to compose an email message. Use `C-c C-c` to send it via the enabled method.

7.3 Mail Aliases

People don't like to have to remember long complex strings of sometimes random data; that's why programmers invented website bookmarks and mail aliases. There's a couple different ways to use them in GNUS. One is to use the Insidious Big Brother Database (perhaps better known as BBDB) to manage your contacts in a small database. Another option is to use the simple mail alias system provided by your `.mailrc` file. Because it's simpler, more portable, and doesn't require code outside the GNU Emacs base, I'm only going to spend time on this second method.

If you've used other mail programs, you may already have some aliases stored in your `.mailrc` file. In case you don't, the syntax is simple: just create a line that takes the form `alias SHORT LONG`, where `SHORT` is the shortened alias name, and where `LONG` is the actual email address. Once you've got that file, you again have two options for using it.

One method is really truly using old-school mail aliases, where the aliases are expanded just before the message is sent. You don't have to do anything extra to enable this — it should happen by default. The second method is somewhat more useful: it treats mail aliases as abbreviations, expanding them automatically when you insert a word-delimiting character like the comma or space. In order to activate this functionality, you have to set a hook for message-mode:

```
(add-hook 'mail-mode-hook 'mail-abbrevs-setup)
```

Now when you're typing in recipient addresses, if you type the name of an alias from your `.mailrc` file and hit comma or space, the alias will be expanded to the address it represents. Much faster than searching through an address book, and certainly much easier than having to remember all those crazy addresses people make up!

7.4 NNTP Troubles

With GNUS 5.11 running under Emacs 22.0.94.1 and Emacs 23.0.0.1 I have a problem regarding my NNTP connection. As far as I can tell, GNUS keeps the connection to the NNTP server open for as long as possible, but the NNTP server will stop paying attention after a while. GNUS doesn't detect this, and when you try to refresh your Usenet subscriptions, GNUS will hang, waiting for the server to send it data, until you eventually C-g it into submission. To automatically kill the hang, set the variable `nntp-connect-timeout` to some value in seconds—when this period of time expires, GNUS will give up.

That doesn't actually solve the problem, though, because the connection is still active, and GNUS will continue to try to get data from this connection. To refresh your Usenet subscriptions, you need to close

the connection to the NNTP server. This can be done either by calling the `nntp-close-server` function, or by entering the server buffer and closing it there. After the server connection is closed, you can re-open it (again, either by the `nntp-open-server` function or through the server buffer) and everything will work again. Until the server lets the connection die again.

```
(setq nntp-connection-timeout 30)
(add-hook 'gnus-after-getting-new-news-hook 'nntp-close-server)
(add-hook 'gnus-started-hook 'nntp-close-server)
```

One decent fix is to use the above server-close method, adding `nntp-close-server` to the `gnus-after-getting-new-news-hook`. This will automatically close the connection after GNUS is done checking the NNTP server; and the next time you go to check for new news, the server will automatically be re-opened. So far, this seems to be an effective workaround for the NNTP connection problem.

8 The Emacs Multimedia System

EMMS is an Emacs-based front-end to play multimedia files. EMMS reads metadata from files and uses that information to list them in the current playlist. Unfortunately, it's a bit slow at reading that metadata, so to increase the speed we can bulk read a bunch of the data ahead of time and store it in EMMS' cache using this Python script I wrote.

8.1 Installation

You can download EMMS using Darcs, a distributed version control system written in Haskell. To fetch the source code, use the following command:

```
darcs get http://www.kollektiv-hamburg.de/~forcer/darcs/emms
```

You'll most likely need to edit the Makefile before you build. If you get errors about `nnheader-concat` not being defined, run this code snippet within the `emms` directory to convert that function call into an equivalent, more widely available alternative:

```
sed -e 's/nnheader-concat \([^ ]+\)/concat (file-name-as-directory \1)/' \
-i.bak `grep -Hl "nnheader-concat" *.el`
```

Now you can do the usual `make` and `sudo make install` goodness. Note that this is fixed in more recent versions.

8.2 Keybindings

```
(global-set-key (kbd "<f1>") 'emms-previous)
(global-set-key (kbd "<f2>") 'emms-start)
(global-set-key (kbd "<f3>") 'emms-stop)
(global-set-key (kbd "<f4>") 'emms-next)
```

8.3 Players

The following I use for defining which players will be used. You'll have to throw the `mplayer` entries in if you want to do things like play Internet radio.

```
(setq emms-player-list '(emms-player-mpg321
                        emms-player-ogg123
                        emms-player-mplayer-playlist
                        emms-player-mplayer))
```

8.4 Modeline Display

I like to see what's playing in the modeline. This function mostly works, only I can't figure out why it disappears when the track changes by itself.

```
(setq emms-mode-line-mode-line-function
  (lambda nil
    (let ((track (emms-playlist-current-selected-track)))
      (let ((title (emms-track-get track 'info-title)))
        (if (not (null title))
            (format emms-mode-line-format title)
            (if (not (null (string-match "^url: " (emms-track-simple-description track))))
                (format emms-mode-line-format "Internet Radio")
                (format emms-mode-line-format "Unknown"))))))))
```

8.4.1 Track Description

This function describes how to display a track entry in the playlist. I like to have mine in the format "Artist: Album - [Num] Title".

```
(setq emms-track-description-function
  (lambda (track)
    (let ((artist (emms-track-get track 'info-artist))
          (album (emms-track-get track 'info-album))
          (number (emms-track-get track 'info-tracknumber))
          (title (emms-track-get track 'info-title)))
      (if (and artist album title)
          (if number
              (format "%s: %s - [%03d] %s" artist album (string-to-int number) title)
              (format "%s: %s - %s" artist album title))
          (emms-track-simple-description track))))))
```

Additionally, we can change the face for the playlist to be a little cooler.

```
(set-face-attribute 'emms-playlist-track-face nil :font "DejaVu Sans-10")
(set-face-attribute 'emms-playlist-selected-face nil :background "White" :foreground "Firebrick")
```

8.5 Adding Tracks from Dired

You can add your own keybinding to Dired to make it add the tracks in a subdirectory when, for example, you press the "e" key. Here's the method to do that:

```
(defun dired-add-to-emms-playlist nil
  "Adds directory tree rooted at the selected directory to the current EMMS playlist."
  (interactive)
  (let ((file (expand-file-name (dired-get-filename))))
    (if (and file (file-directory-p file))
        (emms-add-directory-tree file)
        (emms-add-file file))))

(add-hook 'dired-mode-hook 'my-dired-mode-hook)

(defun my-dired-mode-hook()
  (define-key dired-mode-map (kbd "e") 'dired-add-to-emms-playlist))
```

Now whenever you're browsing your music hierarchy in Dired you can press the "e" key when the cursor is over a directory to add that entire directory's (and subdirectories') contents to the playlist. If you press "e" when over a regular file, that file will be added to the playlist.

8.6 Metadata Loading

This code is my own custom metadata loader. I wrote a Python script which examines a file and gathers the metadata for that file. This I found much more convenient, accurate, and generally faster than the loaders used by EMMS by default. Plus I could easily refactor this into a cache preloader.

```
(defun metadata (track)
  (when (eq 'file (emms-track-type track))
    (with-temp-buffer
      (when (zerop
            (apply (if (fboundp 'emms-i18n-call-process-simple)
                      'emms-i18n-call-process-simple
                      'call-process)
                  "metadata.py"
                  nil t nil
                  (list (emms-track-name track))))
          (goto-char (point-min))
          (while (looking-at "^\\([^\n]+\)=\\((.*\\)$")
                (let ((name (intern (match-string 1)))
                    (value (match-string 2)))
                  (when (> (length value)
                          0)
                    (emms-track-set track
                                     name
                                     (if (eq name 'info-playing-time)
                                         (string-to-number value)
                                         value))))
                  (forward-line 1))))))
  (setq emms-info-functions '(metadata)))
```

Part II

Implementation

Here is the actual source code of my Emacs configuration.

```
12 (emacs 12)≡
    <Pre-Load Variables 13a>
    <Major Mode Selection 14a>
    <Packages 14c>
    <Other Libraries 15a>
    <Personal Variables 16c>
    <Global Configuration 17b>
    <Global Faces 19c>
    <Global Key Bindings 19a>
    <Utility Functions 20c>
    <File Management 21b>
    <Printing 23f>
    <Major Mode Customization 24c>
    <Server 26e>
```

9 Pre-Load Setup

Here we configure the load path and other variables that affect how libraries are loaded.

13a *(Pre-Load Variables 13a)*≡ (12)
(Load Path 13b)
(Load Control Variables 13c)
(Info Directories 13d)

9.1 Load Path

I've got quite a lot of Emacs Lisp files scattered throughout several directories, so I will set them in the load path here. Some of these things I don't even use much anymore.

13b *(Load Path 13b)*≡ (13a)

```
(add-to-list 'load-path "~/elisp")
(add-to-list 'load-path "~/elisp/asn1")
(add-to-list 'load-path "~/elisp/bbdb")
(add-to-list 'load-path "~/elisp/haskell")
(add-to-list 'load-path "~/elisp/logo")
(add-to-list 'load-path "~/elisp/muse")
(add-to-list 'load-path "~/elisp/planner")
(add-to-list 'load-path "~/elisp/remember")
(add-to-list 'load-path "~/elisp/rcirc")
(add-to-list 'load-path "~/elisp/slime")
(add-to-list 'load-path "~/elisp/sml")
(add-to-list 'load-path "~/elisp/swank-clojure")
```

9.2 Conditional Load Variables

Nowadays I don't use EMMS or GNUS so there's no reason to load them. However, I do use rcirc because it's the only good IRC client for Mac (you heard me). I keep the load control and configuration of these larger pieces in separate files, to help make it easier to conditionally load them. These will be checked later in the `[[Other Libraries]]` section.

13c *(Load Control Variables 13c)*≡ (13a)

```
(setq tcv-emms-enable nil)
(setq tcv-gnus-enable nil)
(setq tcv-rcirc-enable t)
```

9.3 Info Directories

In order to get good use out of the info browser, you need to tell it where the info files are for the Emacs Lisp code you have installed. The system default info directories will get picked up automatically.

13d *(Info Directories 13d)*≡ (13a)

```
(add-to-list 'Info-default-directory-list "~/elisp/bbdb/info")
(add-to-list 'Info-default-directory-list "~/elisp/logo/info")
```

10 Major Mode Setup

This is a mechanism for loading modes automatically when certain files are opened, and for making the association that certain files should use specific major modes. It operates on a data structure that describes the relationship between file names and modes.

Each entry in the list that we process can have one of two forms. In the first, the fields in the list are:

1. A function which starts the desired mode, for example "lua-mode" to start Lua mode.
2. The longer descriptive name of the mode, which is used by Emacs somehow.
3. Regular expression that the file name is matched against to determine when this mode should be used. Continuing with the Lua example, this would be "\\ .lua\$".

This case assumes that the Elisp file that contains the definition of the specified function to load the mode has the same name as that function. This is typically the case by convention: the file `lua-mode.el` contains the definition of `lua-mode`. However, sometimes this is not the case, for example Prolog support comes in a file called `prolog.el` and so we offer the alternate method. If the first element of the entry is a list, rather than a string, then it contains these items:

1. The name of the function to start the mode, for example “`prolog-mode`”.
2. The name of the file which includes the definition of that function, without extension, e.g. “`prolog`”.

The rest is as above.

```
14a (Major Mode Selection 14a)≡ (12)
(mapcar
  (lambda (mode-list)
    (if (listp (car mode-list))
        (progn
          (autoload (car (car mode-list)) (cadr (car mode-list))
                    (concat "Major mode for editing " (cadr mode-list) "source code. "))
          (add-to-list 'auto-mode-alist '(, (cadr (cdr mode-list)) . ,(car (car mode-list)))))
        (progn
          (autoload (car mode-list) (symbol-name (car mode-list))
                    (concat "Major mode for editing " (cadr mode-list) "source code. "))
          (add-to-list 'auto-mode-alist '(, (cadr (cdr mode-list)) . ,(car mode-list)))))
    '( (asn1-mode      "ASN.1"          "\\ .asn$")
      (clojure-mode   "Clojure"         "\\ .clj$")
      (css-mode       "Cascading Style Sheets" "\\ .css$")
      (erlang-mode    "YAWS Dynamic Page" "\\ .yaws$")
      (fvwm-mode      "FVWM Configuration" "\\ .fvwm/config$")
      (haskell-mode   "Haskell"         "\\ .hs$")
      (lua-mode       "Lua"             "\\ .lua$")
      (cperl-mode     "Perl"            "\\ .p[1m]$")
      (php-mode       "PHP"             "\\ .php\\|\\.inc$")
      (python-mode    "Python"         "\\ .[j]p[y]$")
      (ruby-mode      "Ruby"           "\\ .rb$")
      (rst-mode       "reStructuredText" "\\ .rst$")
      (smalltalk-mode "Smalltalk"       "\\ .st$")
      (sml-mode       "Standard ML"     "\\ .sml$")
      (markdown-mode  "Markdown"        "\\ .text$")
      (wikipedia-mode "Wikipedia"       "\\ .wiki$")
      ((js2-mode      "js2")            "JavaScript" "\\ .js$")
      ((logo-mode     "logo")           "Logo"        "\\ .logo$")
      ((org-mode      "org")            "Org"         "\\ .org$")
      ((prolog-mode   "prolog")         "Prolog"      "\\ .pdb$")
      ((tuareg-mode   "tuareg")         "Caml"        "\\ .ml$"))))
```

We also have some more old-school loading going on.

```
14b (Major Mode Selection 14a)+≡ (12)
(add-to-list 'auto-mode-alist '("Makefile" . makefile-gmake-mode))
(add-to-list 'auto-mode-alist '("JSP" . html-mode))
```

11 Other Packages to Load

```
14c (Packages 14c)≡ (12)
(require 'autoinsert)      ; insert text when opening new files
(require 'calendar)        ; organize your life
(require 'compile)         ; display the results of compiling a file
(require 'cperl-mode)     ; a superior Perl mode
(require 'darcsum)         ; finds the differences in Darcs repositories
(require 'doxymacs)        ; Doxygen integration
(require 'erc)             ; the ERC chat client
(require 'erlang-start)    ; support for Erlang
(require 'faraday-mode)    ; electricize arbitrary text
(require 'htmlize)         ; convert a buffer into HTML
(require 'psvn)            ; enhanced Subversion interaction
(require 'remember-planner) ; I don't remember what this does! :)
(require 'rst)             ; reStructuredText mode
```

```
(require 'quack)                ; superior mode for Scheme programming
(require 'sml-autoloads)       ; autoloads necessary SML mode functions
(require 'speedbar)            ; file and tag browser
(require 'stamp)               ; set timestamps in files
```

Some files are written to be better simply loaded than required in some other more complex sense.

15a

(*Other Libraries* 15a)≡

(12)

```
(if tcv-emms-enable (load "-/.emacs_emms.el"))
(if tcv-gnus-enable (load "-/.emacs_gnus.el"))
(if tcv-rcirc-enable (load "-/.emacs_rcirc.el"))
(load "~/elisp/haskell/haskell-site-file") ; load haskell stuff
(setq haskell-program-name "/opt/haskell/bin/ghci")
<AUCTeX 16a>
<SLIME 16b>

(eval-after-load "htmlize"
  '(progn
    (defadvice htmlize-faces-in-buffer (after org-no-nil-faces activate)
      "Make sure there are no nil faces"
      (setq ad-return-value (delq nil ad-return-value))))))

;; Emacs Muse

(require 'muse-mode)
(require 'muse-html)
(require 'muse-latex)
(require 'muse-project)

;(autoload 'imaxima "imaxima" "Image support for Maxima." t)

;; Set the action table for faraday electric mode.

(setq faraday-action-table
  '((sml-mode . (("type" . indent-according-to-mode)
                ("val" . indent-according-to-mode))))))

;; Variables specific to external packages

(setq erc-fill-column 100)
(setq erc-nick "metasyntax|work")

(setq erlang-root-dir "/usr/local/lib/erlang")
(setq js2-auto-indent-flag nil)

(setq quack-global-menu-p nil)
(setq quack-fontify-style 'emacs)
(setq quack-default-program "gsi")

(setq remember-handler-functions '(remember-planner-append))
(setq remember-annotation-functions planner-annotation-functions)
(setq rst-mode-lazy nil)          ; fix horribly slow font-locking in reStructuredText

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; PLANNER
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq muse-project-alist
  '(("Project"
    ("~/emacs-muse" :default "index")
    (:base "html" :path "~/public_html")
    (:base "pdf" :path "~/public_html/pdf"))))

15b (UC Berkeley Logo 15b)≡
  (setq logo-system-type 'xterm)
  (setq logo-binary-name "/usr/local/bin/logo")
  (setq logo-info-file "~/elisp/info/ucblog.info")
```

```
(setq logo-help-path "/usr/local/share/ucblog/helpfiles/")
(setq logo-tutorial-path "~/elisp/logo")
(setq logo-setcursor-overwrite nil)
(setq logo-unbalanced-distance 4096)
(setq logo-info-trans nil)
```

```
16a <AUCTeX 16a>≡ (15a)
(load "auctex.el" t t t) ; load AUCTeX
(load "preview-latex.el" t t t) ; load preview-latex
(setq LaTeX-verbatim-environments '("verbatim" "verbatim*" "Verbatim"))
(setq LaTeX-command-style '((" " /usr/local/texlive/2009/bin/x86_64-linux/%(PDF)%(latex) %S%(PDFout))))
```

```
16b <SLIME 16b>≡ (15a)
(require 'slime) ; autoloads a superior Lisp interaction mode
(require 'swank-clojure) ; to use Clojure from within SLIME

(setenv "SBCL_HOME" "/opt/sbcl/lib/sbcl")
(setq inferior-lisp-program "/opt/clisp/bin/clisp")
(slime-setup '(slime-repl))

;; This is to use Clojure from within SLIME.

(setq swank-clojure-jar-path "~/Libraries/Java/clojure.jar")
(setq swank-clojure-extra-classpaths nil)
(require 'swank-clojure-autoload)
```

12 Personal Variables

These are variables that are used by my own functions later on. The bulk of this is font definition and so forth, but there are a few other things as well.

```
16c <Personal Variables 16c>≡ (12)
<X11 Fonts 16d>
<TrueType Fonts 17a>

(defvar tcv-x-font tcv-swiss
  "Which X font to use.")

(defvar tcv-xft-font (if (string= system-type "darwin") "mac" "dejavu"))

(defvar tcv-use-x-font-p (if (string= system-name "darwin") nil nil)
  "Use a regular X font (as opposed to Xft font) when true.")

(defvar tcv-browser 'firefox
  "Type of web browser to use. Sets browser functions in Emacs
  appropriately. Currently supported: firefox, conkeror, opera.")

(defvar tcv-every-mode '(ada-mode c-mode c++-mode clojure-mode
  cperl-mode css-mode emacs-lisp-mode erlang-mode
  java-mode haskell-mode html-mode lisp-mode perl-mode
  php-mode prolog-mode ruby-mode scheme-mode sgml-mode
  sh-mode shell-script-mode sml-mode tcl-mode
  tuareg-mode xml-mode)
  "Every single mode.")
```

12.1 X11 Fonts

I still like to use bitmapped fonts on my Unix systems sometimes. These are a variety of useful fonts; set `tcv-x-font` to one of these variables to use that font.

```
16d <X11 Fonts 16d>≡ (16c)
(defvar tcv-lispm-fixed "-lispm-fixed-medium-r-normal--13-90-100-100-m-80-iso8859-15"
  "X font designator for the MIT CADR Lisp Machine type.")

(defvar tcv-dec-terminal "-dec-terminal-bold-r-normal--14-140-75-75-c-80-iso8859-1"
```

```

"X font designator for the bolded DEC terminal type.")

(defvar tcv-swiss "-xos4-terminus-medium-r-normal--16-160-72-72-c-80-iso10646-1"
  "X font designator for my modified Terminus font.")

(defvar tcv-misc-fixed-large "-misc-fixed-medium-r-normal--13-120-75-75-c-70-iso10646-1"
  "Miscellaneous fixed Unicode font - larger version.")

(defvar tcv-misc-fixed-small "-misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso10646-1"
  "Miscellaneous fixed Unicode font - smaller version.")

```

12.2 TrueType Fonts

Register some common TrueType families. This makes it so we can choose a “theme” for the entire program, and then load the default families for each typeface in their appropriate roles. For example, we can use all DejaVu families, or the standard Mac families.

Each font that is defined should have three fields, which are used by `make-xft-font` to create a string that Emacs will use to load that family.

```

17a <TrueType Fonts 17a>≡ (16c)
  (defvar tcv-font-droid
    '(:sans . "Droid Sans"
      (:serif . "Droid Serif")
      (:mono . "Droid Sans Mono")))

  (defvar tcv-font-dejavu
    '(:sans . "DejaVu Sans"
      (:serif . "DejaVu Serif")
      (:mono . "DejaVu Sans Mono")))

  (defvar tcv-font-liberation
    '(:sans . "Liberation Sans"
      (:serif . "Liberation Serif")
      (:mono . "Liberation Mono")))

  (defvar tcv-font-mac
    '(:sans . "Helvetica"
      (:serif . "Times")
      (:mono . "Monaco")))

  (defun make-xft-font (family variant size)
    (concat (cdr (assoc variant (eval (intern (concat "tcv-font-" family)))))) "-" (int-to-string size)))

17b <Global Configuration 17b>≡ (12)
  <Frame Setup 17c>
  <Browser Setup 18a>
  <Global Variables 18b>
  <Buffer-Local Variables 18c>
  <Set By Function 18d>

```

We set the information that each frame will use: dimensions, font backend (somewhat legacy from the days when XFT was not enabled by default), and the face. It was once the case that if we failed to set the face here it used the Emacs-internal default for new frames.

```

17c <Frame Setup 17c>≡ (17b)
  (setq default-frame-alist '((width . 80) (height . 40)
                             (font-backend . "xft, x")
                             ,(if tcv-use-x-font-p
                                 '(font . ,tcv-x-font))))
  ;(setq initial-frame-alist '((width . 120) (height . 40)))

```

We have a set of rules for how we determine what browser to use, which comes down to OS and how the variable above is set for which browser we would *like* to use.

```
18a (Browser Setup 18a)≡ (17b)
  (cond
    ((eql tcv-browser 'firefox)
     (setq browse-url-new-window-flag t)
     (setq browse-url-firefox-new-window-is-tab t)
     (setq browse-url-browser-function 'browse-url-firefox))
    ((eql tcv-browser 'conkeror)
     (setq browse-url-new-window-flag nil)
     (setq browse-url-firefox-program "/opt/conkeror/xulrunner-stub")
     (setq browse-url-browser-function 'browse-url-firefox))
    ((eql tcv-browser 'opera)
     (setq browse-url-generic-program "opera")
     (setq browse-url-generic-args '("-newpage"))))
```

These variables can be set globally.

```
18b (Global Variables 18b)≡ (17b)
  (setq c-basic-offset 4) ; set indentation to four spaces
  (setq case-fold-search t) ; don't remember what this does
  (setq column-number-mode t) ; show column numbers on the modeline
  (setq default-input-method "TeX") ; make TeX the alternate input method
  (setq dired-recursive-deletes 'always) ; don't ask to delete recursively
  (setq display-time-format "%Y-%m-%d %H:%M") ; use international format for time

  (setq frame-title-format '(:eval (if (null (buffer-file-name))
                                       "%b"
                                       "%f"))))

  (setq inhibit-startup-message t) ; don't show the Emacs splash buffer
  (setq make-backup-files nil) ; stop making those silly backup files
  (setq mouse-wheel-follow-mouse t) ; mouse wheel affects window over which mouse hovers
  (setq mouse-wheel-progressive-speed nil) ; don't accelerate the mouse wheel
  (setq require-final-newline t) ; automatically put newline at end of file
  (setq scheme-program-name "csi") ; Chicken Scheme Interpreter / crime scene investigation
  (setq sql-sqlite-program "sqlite3") ; modern version of sqlite program
  (setq tcl-application "tclsh") ; Tcl program to run in inferior mode
  (setq transient-mark-mode t) ; don't remember what this does
  (setq truncate-partial-width-windows nil) ; ... ?
  (setq uniquify-buffer-name-style 'forward) ; ... ?
  (setq use-dialog-box nil) ; don't use the GUI for questions / file finding / etc.
  (setq user-full-name "Taylor Venable") ; full name, in case not set
  (setq visible-bell t) ; flash the top and bottom lines instead of beeping
  (setq vc-follow-symlinks t) ; don't ask to follow symlinks to VC files
  (setq x-stretch-cursor t) ; stretch cursor to cover tabs

  ; (setq sgml-quick-keys t) ; make SGML tag characters behave electrically
  ; (setq mouse-wheel-progressive-speed t) ; enable fast mouse scrolling
```

These variables are usually buffer-local and so we have to use the special `setq-default` function to set their default global values (which then become buffer-local if they are changed).

```
18c (Buffer-Local Variables 18c)≡ (17b)
  (setq-default fill-column 80) ; fill up to column 80
  (setq-default indent-tabs-mode nil) ; never use tabs - only spaces
  (setq-default indicate-empty-lines t) ; show empty lines in the fringe

  ; (setq-default show-trailing-whitespace t) ; highlight trailing whitespace
```

Finally some variables cannot be directly set for some reason, or it is recommended that they not be set directly. These we set by using the function.

```
18d (Set By Function 18d)≡ (17b)
  (delete-selection-mode t) ; delete text in marked region
  (global-font-lock-mode t) ; enable syntax highlighting
  (mouse-wheel-mode t) ; allow scrolling with mouse wheel
  (set-language-environment "UTF-8") ; use UTF-8 for default input
  (set-scroll-bar-mode 'right) ; show scrollbar on the right
  (show-paren-mode t) ; highlight matching parentheses
```

```
(tool-bar-mode -1)                ; turn off quick tool bar
(display-time-mode)              ; show time in the modeline

(grep-compute-defaults)
(setq grep-command "grep -rnHE --exclude-dir='.svn' --exclude='*/TAGS'")
```

```
19a (Global Key Bindings 19a)≡ (12)
(global-set-key (kbd "<f1>") 'emms-previous)
(global-set-key (kbd "<f2>") 'emms-start)
(global-set-key (kbd "<f3>") 'emms-stop)
(global-set-key (kbd "<f4>") 'emms-next)

(global-set-key (kbd "C-,") 'hs-toggle-hiding)
(Darwin Key Bindings 19b)
```

These key bindings emulate the typical bindings that you find in OS X. We have to work around several things here:

- The delete key does backspace functions.
- Home and end goto the beginning and end of the document.

```
19b (Darwin Key Bindings 19b)≡ (19a)
(cond ((string= system-type "darwin")
      (progn
        (global-set-key (kbd "<s-down>") #'end-of-buffer)
        (global-set-key (kbd "<s-up>") #'beginning-of-buffer)
        (global-set-key (kbd "<M-kp-delete>") #'backward-kill-word)
        (global-set-key (kbd "<kp-delete>") #'delete-char)
        (global-set-key (kbd "<M-kp-delete>") #'kill-word)
        (global-set-key (kbd "<s-left>") #'beginning-of-line)
        (global-set-key (kbd "<s-right>") #'end-of-line)
        (global-set-key (kbd "<home>") #'beginning-of-line)
        (global-set-key (kbd "<end>") #'end-of-line)
        (global-set-key (kbd "<s-s>") #'save-buffer)
        (global-set-key (kbd "<s-q>") #'save-buffers-kill-terminal)
        (global-set-key (kbd "<s-w>") #'kill-buffer))))))
```

```
19c (Global Faces 19c)≡ (12)
(Adjust Face Function 19d)
(Customized Faces 19e)
(Diff Faces 20a)
(Custom Word Highlight 20b)
```

```
19d (Adjust Face Function 19d)≡ (19c)
(defun tcv-adjust-font ()
  (if tcv-use-x-font-p
      (progn
        (set-face-attribute 'default nil :font tcv-x-font)
        (set-face-attribute 'tooltip nil :font tcv-x-font)
        (set-face-attribute 'mode-line-inactive nil :font tcv-x-font))
      (let ((font-size (cond ((string-match "^lionel" system-name) 9.5)
                            ((string= "darwin" system-type) 13)
                            (t 11))))
        (set-face-attribute 'default nil :font (make-xft-font tcv-xft-font :mono font-size))
        (set-face-attribute 'fixed-pitch nil :font (make-xft-font tcv-xft-font :mono font-size))
        (set-face-attribute 'variable-pitch nil :font (make-xft-font tcv-xft-font :sans font-size))
        (set-face-attribute 'tooltip nil :weight 'medium)
        (set-face-attribute 'font-lock-doc-face nil :slant 'italic))))

  (if (and (display-graphic-p)
          (not (boundp 'aquamacs-version)))
      (tcv-adjust-font)))
```

```
19e (Customized Faces 19e)≡ (19c)
(set-face-attribute 'cursor nil :background "Gray50")
(set-face-attribute 'font-lock-comment-face nil :foreground "Firebrick")
(set-face-attribute 'font-lock-doc-face nil :foreground "Firebrick")
(set-face-attribute 'font-lock-string-face nil :foreground "RoyalBlue")
(set-face-attribute 'highlight nil :background "LightSteelBlue")
```

```
(set-face-attribute 'mode-line          nil :background "gray75" :box nil)
(set-face-attribute 'mode-line-highlight nil :box nil :foreground "Firebrick")
(set-face-attribute 'mode-line-inactive nil :background "grey90" :box nil)
(set-face-attribute 'region             nil :background "LemonChiffon2")
(set-face-attribute 'show-paren-match   nil :background "SkyBlue3" :foreground "White")
(set-face-attribute 'show-paren-mismatch nil :background "IndianRed3" :foreground "White")
```

20a *(Diff Faces 20a)*≡ (19c)

```
(require 'diff-mode)
(set-face-attribute 'diff-removed      nil :background "RosyBrown1")
(set-face-attribute 'diff-added        nil :background "PaleGreen1")
(setq-default diff-switches "-u")
```

20b *(Custom Word Highlight 20b)*≡ (19c)

```
(make-face 'tcv-special-words)
(set-face-attribute 'tcv-special-words nil :foreground "White" :background "Firebrick")

(let ((pattern "\\<\\(\\(FIXME\\|TODO\\|NOTE\\|WARNING\\|NOTREACHED\\):"))
      (mapcar
       (lambda (mode)
         (font-lock-add-keywords mode '((,pattern 1 'tcv-special-words prepend))) tcv-every-mode))
```

13 Utility Functions

20c *(Utility Functions 20c)*≡ (12)

```
(Insert Date Function 20d)
(Hex Decode Function 20e)
(Longest Line Function 20f)
(Protected Require Function 21a)
```

20d *(Insert Date Function 20d)*≡ (20c)

```
(defun insert-date-rfc822 ()
  "Insert the current time in the format used by RFC-822."
  (interactive)
  (insert (format-time-string "%a, %d %b %Y %H:%M:%S %Z")))
```

13.1 Hex Decoding

20e *(Hex Decode Function 20e)*≡ (20c)

```
(defun tcv-decode-hex ()
  "Turn a series of hex characters (like in APDU logs) into ASCII."
  (interactive)
  (if (not (region-active-p))
      (error "You must select a region")
      (progn
        (if (< (mark) (point))
            (exchange-point-and-mark))
        (insert " ASCII: ")
        (while (re-search-forward "\\([0-9A-F]\\{2\\}\\)" (mark) t)
          (let ((code-as-decimal (string-to-number (match-string 1) 16)))
            (if (and (> code-as-decimal 31) (< code-as-decimal 127))
                (replace-match (format "%c" (make-char 'ascii code-as-decimal)) nil nil))))))
```

13.2 Longest Line

20f *(Longest Line Function 20f)*≡ (20c)

```
(defun longest-line-length nil
  (let ((longest-line 0)
        (line 0)
        (length 0))
    (save-excursion
      (goto-char (point-min))
      (end-of-line)
      (setq length (current-column))
```

```

    (while (zerop (forward-line 1))
      (setq line (1+ line))
      (end-of-line)
      (cond ((> (current-column) length)
             (setq length (current-column))
             (setq longest-line line))))
    length))

```

13.3 Protected Require

21a *(Protected Require Function 21a)*≡ (20c)

```

(defun protected-require (name)
  (condition-case err
    (progn
      (if (stringp name)
          (load-library name)
          (require name)))
      (file-error nil)))

```

14 File Management

21b *(File Management 21b)*≡ (12)

```

<Code Boilerplates 21c>
<Make Scripts Executable 22a>
<Work Coding Conventions 22b>
<Path Functions 22c>
<Find Changes File 23b>
<Add Changes Entry 23c>

```

14.1 Boilerplate & Templates

21c *(Code Boilerplates 21c)*≡ (21b)

```

(setq auto-insert-directory "~/Templates/") ; where the templates are stored
(setq auto-insert-query nil) ; don't ask to insert -- just do it
(setq latex-do-class-assignment-insert nil) ; avoid screwing up AUCTeX

;; Prompt for and expand CLASS and ASSIGNMENT in the LaTeX template.

(defun latex-insert-class-assignment nil
  (if latex-do-class-assignment-insert
      (let ((class (read-from-minibuffer "Class: "))
            (assignment (read-from-minibuffer "Assignment: ")))
        (progn
          (while (search-forward "CLASS" nil t)
            (replace-match class t t))
          (while (search-forward "ASSIGNMENT" nil t)
            (replace-match assignment t t))
          (setq latex-do-class-assignment-insert nil)))
      nil))

;; Returns a list associating filename regexes to the template file.
;; For LaTeX, also add the CLASS and ASSIGNMENT expansion function.

(defun template-list (insert-directory)
  (let (lst)
    (dolist (elt (directory-files insert-directory nil "[^\\.]" nil) lst)
      (if (string-equal elt "tex")
          (setq lst (append lst '((, (concat "\\." elt "$") . [ ,elt latex-insert-class-assignment])))
                (setq lst (append lst '((, (concat "\\." elt "$") . ,elt)))))))

    (setq auto-insert-alist (template-list auto-insert-directory))
    (auto-insert-mode)

```

```
;; This makes it so we only do the class & assignment stuff when creating
;; a new file. If we enable it all the time, AUCTeX gets messed up.

(add-hook 'find-file-not-found-functions
  (lambda nil
    (if (string-equal (file-name-extension buffer-file-name) "tex")
        (setq latex-do-class-assignment-insert t)
        nil)))
```

14.2 Executable Scripts

22a *(Make Scripts Executable 22a)*≡ (21b)

```
(add-hook 'after-save-hook
  (lambda nil
    (and (save-excursion
          (save-restriction
            (widen)
            (goto-char (point-min))
            (save-match-data
              (looking-at "~#!"))))
         (not (file-executable-p buffer-file-name))
         (shell-command (concat "chmod +x " (shell-quote-argument buffer-file-name)))
         (message
          (concat "Saved as executable script: " buffer-file-name))))))
```

14.3 Coding Conventions

22b *(Work Coding Conventions 22b)*≡ (21b)

```
(defun work-code-conventions ()
  "Configure Emacs variables to support work coding conventions.
Intended to be used as a hook for finding a file."
  (if (string-match "Work/" buffer-file-name)
      (progn
        (set-variable 'indent-tabs-mode nil)
        (set-variable 'tab-width 4))))

(add-hook 'find-file-hook 'work-code-conventions)
(add-hook 'before-save-hook 'delete-trailing-whitespace)

; I just figured out how input methods work and they're too damn useful.
; (global-set-key (kbd "C-\\") #'hs-toggle-hiding)

(require 'cl)
```

14.4 Path Functions

Surprisingly, there are no functions in Emacs Lisp to manipulate path components. This is doubly confusing given that Common Lisp did a great job practically inventing logical paths. Thus, we are forced to write our own path manipulation functions here, assuming that paths uses forward slashes as directory separators.

22c *(Path Functions 22c)*≡ (21b)
(Path Split 22d)
(Path Parent 23a)

22d *(Path Split 22d)*≡ (22c)

```
(defun path-split (path)
  (split-string path "/" t))
```

23a *(Path Parent 23a)*≡ (22c)

```
(defun path-parent (path)
  (let ((components (path-split path)))
    (if (= (length components) 1) "/"
        (concat "/" (reduce #'(lambda (old new)
                               (concat old "/" new))
                             (reverse (cdr (reverse components))))))))
```

14.5 Find Changes File

23b *(Find Changes File 23b)*≡ (21b)

```
(defun tcv-find-changes-file (path)
  (let ((try (concat (file-name-directory path) "CHANGES")))
    (if (file-exists-p try)
        try
        (if (string= (path-parent try) "/")
            nil
            (tcv-find-changes-file (path-parent try))))))
```

14.6 Add Changes Entry

23c *(Add Changes Entry 23c)*≡ (21b)

```
(setq-default add-log-full-name "Taylor Venable")
(setq-default add-log-mailing-address "taylor.venable@trustbearer.com")

(defun tcv-add-changes ()
  (interactive)
  (let ((changes-file (or (tcv-find-changes-file (buffer-file-name)) "CHANGES")))
    (add-change-log-entry nil changes-file nil t)))
```

23d *(Speedbar 23d)*≡

```
(setq speedbar-frame-parameters
      '((minibuffer)
        (width . 36)
        (height . 50)
        (border-width . 0)
        (menu-bar-lines . 1)
        (tool-bar-lines . 0)
        (unsplittable . t)
        (left . 938)
        (top . 47)))

(setq speedbar-indentation-width 2) ; use two spaces to indent sub-files
(setq speedbar-show-unknown-files t) ; show files we don't have tags for

;; (setq speedbar-use-imenu-flag nil)
;; (setq speedbar-fetch-etags-command "/usr/bin/ctags-exuberant")
;; (setq speedbar-fetch-etags-arguments '("-e" "-f" "-"))
;; (speedbar)
```

23e *(Calendar 23e)*≡

```
(setq calendar-date-display-form
      '((format "%4s-%02d-%02d" year (string-to-int month) (string-to-int day))))
(add-to-list 'diary-date-forms '(year "-" month "-" day))
```

23f *(Printing 23f)*≡ (12)

```
(setq ps-mark-code-directory nil) ; don't know what this does, but it's necessary
(setq ps-font-size '(10 . 10)) ; make the font actually legible
(setq ps-current-bg nil) ; set the background color of the output
(setq ps-print-color-p nil) ; print only black (for mono laser printer)

(setq ps-right-header
      (list "/pagenumberstring load"
            'ps-time-stamp-yyyy-mm-dd
            'ps-time-stamp-hh:mm:ss))
```

(a2ps Function 24a)

```

24a <a2ps Function 24a>≡ (23f)
      (defun a2ps-print nil
        (interactive)
        (shell-command (concat "a2ps "
                               "--medium=Letter "
                               (if (> (longest-line-length) 95) "--landscape " "--portrait ")
                               "--columns=1 "
                               "--line-numbers=1 "
                               "--header=" " "
                               "--left-title="
                               (shell-quote-argument
                                (format-time-string
                                 "%Y-%m-%d %H:%M:%S"
                                 (nth 5 (file-attributes buffer-file-name))))
                               " "
                               "--left-footer=" " "
                               "--footer=" (shell-quote-argument buffer-file-name) " "
                               "--right-footer=" " "
                               "--font-size=9 "
                               "--tabsize=2 "
                               "--pretty-print "
                               "--highlight-level=heavy "
                               "--sides=duplex "
                               (shell-quote-argument buffer-file-name))))))

```

Ant compilation is now configured by default, so there's no real reason to enable this anymore.

```

24b <Ant Compilation 24b>≡
      ;; (setq compilation-error-regexp-alist
      ;;       (append (list
      ;;                 ;; works for jikes
      ;;                 '(("^\\s-*\\[[^]]*\\]\\s-*\\(.+\\):\\([0-9]+\\):\\([0-9]+\\):[0-9]+:[0-9]+:" 1 2 3)
      ;;                 ;; works for javac
      ;;                 '(("^\\s-*\\[[^]]*\\]\\s-*\\(.+\\):\\([0-9]+\\):" 1 2))
      ;;                 compilation-error-regexp-alist))

```

15 Major Mode Customization

Here we put code for tweaking individual major modes. Usually this means adding hooks to the modes themselves, or otherwise setting buffer-local variables when we enter a buffer with the specific mode.

```

24c <Major Mode Customization 24c>≡ (12)
      <All Modes 24d>
      <C Mode 25b>
      <Java Mode 25c>
      <Haskell Mode 25d>
      <HTML Mode 25e>
      <LaTeX Mode 26a>
      <Python Mode 26b>
      <Ruby Mode 26c>
      <Org Mode 26d>

```

15.1 All Modes

There are a few things we need to define for nearly all modes, which is why we set `tcv-every-mode` above. The first thing we want to do is redefine the binding for the return key to add a new line and then indent according to the current mode. Actually we want to do this in every mode except for Haskell.

```

24d <All Modes 24d>≡ (24c)
      (mapcar
       (lambda (mode)
         (let ((mode-hook (intern (concat (symbol-name mode) "-hook"))))
           (add-hook mode-hook (lambda nil (local-set-key (kbd "RET") 'newline-and-indent))))
         (remove 'haskell-mode tcv-every-mode)))

```

We also create a nifty little function to set the indentation for specific modes. Because some modes define their own indentation variables rather than using `c-basic-offset` we have to set them like this. The function processes a data structure where each element is a list of two things:

1. The hook to attach the indentation adjustment to.
2. The variable that controls the indentation level.

We then go through each one and create a hook function which adjusts the indentation variable, setting it to 4. This is currently hardcoded; there's no facility to use e.g. `2 insetad`.

```
25a <All Modes 24d>+≡ (24c)
      (mapcar
        (lambda (mode-hook-list)
          (add-hook (car mode-hook-list) '(lambda nil (setq ,(cadr mode-hook-list) 4))))
        '((ada-mode-hook      ada-indent)
          (cperl-mode-hook   cperl-indent-level)
          (fortran-mode-hook fortran-do-indent)
          (fortran-mode-hook fortran-if-indent)
          (fortran-mode-hook fortran-structure-indent)
          (ruby-mode-hook    ruby-indent-level)))
```

15.2 C Mode

- Enable hungry delete at the beginning of the line
- Enable sub-word mode.

```
25b <C Mode 25b>≡ (24c)
      (add-hook 'c-mode-hook
        (lambda nil
          (progn
            (c-toggle-hungry-state 1)
            (c-subword-mode))))
```

15.3 Java Mode

- Turn on proper annotation indentation.
- Enable sub-word mode for movement and deletion.

```
25c <Java Mode 25c>≡ (24c)
      (require 'java-mode-indent-annotations)

      (add-hook 'java-mode-hook
        (lambda nil
          (progn
            (java-mode-indent-annotations-setup)
            (c-subword-mode))))
```

15.4 Haskell Mode

- Enable intelligent indentation.

```
25d <Haskell Mode 25d>≡ (24c)
      (add-hook 'haskell-mode-hook 'turn-on-haskell-indent)
```

15.5 HTML Mode

- Set a new regexp to match for time stamping.

```
25e <HTML Mode 25e>≡ (24c)
      (add-hook 'html-mode-hook
        (lambda nil
          (progn
            (set-variable 'stamp-prefix "<div id=\"updated\">" t)
            (set-variable 'stamp-suffix "</div>" t))))
```

15.6 \LaTeX Mode

- Set auto-fill mode.
- Set the compiler to be pdf \LaTeX

```
26a <LaTeX Mode 26a>≡ (24c)
      (add-hook 'latex-mode-hook
        (lambda nil
          (progn
            (auto-fill-mode 1)
            (set-variable 'compile-command "pdflatex" t))))
```

15.7 Python Mode

- Rescan the imenu when we open a file.
- Select different colorization for builtins and keywords.

```
26b <Python Mode 26b>≡ (24c)
      (add-hook 'python-mode-hook
        (lambda nil
          (progn
            (imenu--menubar-select imenu--rescan-item)
            (set-face-attribute 'py-builtins-face nil :foreground "DarkOrange2")
            (set-face-attribute 'py-pseudo-keyword-face nil :foreground "DarkOrange2")
            (setq python-honour-comment-indentation t))))
```

15.8 Ruby Mode

- Add the imenu to Emacs' menu bar.
- Rescane the imenu when we open a file.

```
26c <Ruby Mode 26c>≡ (24c)
      (add-hook 'ruby-mode-hook
        (lambda nil
          (progn
            (imenu-add-to-menubar "Imenu")
            (imenu--menubar-select imenu--rescan-item))))
```

15.9 Org Mode

```
26d <Org Mode 26d>≡ (24c)
      (define-key global-map "\C-cl" 'org-store-link)
      (define-key global-map "\C-ca" 'org-agenda)
      (setq org-export-html-style "<link rel='stylesheet' type='text/css' href='todo.css'/>")
      (setq org-export-html-style-include-default nil)
      (setq org-log-done t)
```

16 Server

Emacs Server allows you to continuously use a single Emacs instance and then load buffers within that instance from an outside process. This is especially useful for opening files from a mail client, web browser, or external file manager (if you're not using dired).

```
26e <Server 26e>≡ (12)
      (add-hook 'server-switch-hook
        (lambda nil
          (let ((server-buf (current-buffer)))
            (bury-buffer)
            (switch-to-buffer-other-frame server-buf))))

      (add-hook 'server-done-hook 'delete-frame) ; delete frame when client done
```

```
(server-start)                ; enable Emacs server

(put 'uppercase-region 'disabled nil)
(put 'downcase-region 'disabled nil)
```

A Colophon

This document was typeset using pdf \TeX , a direct-to-PDF implementation of \TeX with micro-typographic extensions. Most of the typefaces used are from the excellent \TeX Gyre collection from GUST: Pagella (based on Palatino) for section headings, Heros (based on Helvetica) for the abstract, and Bonum (based on Bookman) for the main text body. The monospace typeface is the standard typewriter of the Latin Modern collection, a project also sponsored by GUST, and an extension of Donald Knuth's original Computer Modern typefaces.