XEmacs Internals Manual

Version 1.1, March 1997

Ben Wing
Martin Buchholz

Version 1.1
March, 1997.

1 A History of Emacs

XEmacs is a powerful, customizable text editor and development environment. It began as Lucid Emacs, which was in turn derived from GNU Emacs, a program written by Richard Stallman of the Free Software Foundation. GNU Emacs dates back to the 1970's, and was modelled after a package called "Emacs", written in 1976, that was a set of macros on top of TECO, an old, old text editor written at MIT on the DEC PDP 10 under one of the earliest time-sharing operating systems, ITS (Incompatible Timesharing System). (ITS dates back well before Unix.) ITS, TECO, and Emacs were products of a group of people at MIT who called themselves "hackers", who shared an idealistic belief system about the free exchange of information and were fanatical in their devotion to and time spent with computers. (The hacker subculture dates back to the late 1950's at MIT and is described in detail in Steven Levy's book *Hackers*. This book also includes a lot of information about Stallman himself and the development of Lisp, a programming language developed at MIT that underlies Emacs.)

1.1 Through Version 18

Although the history of the early versions of GNU Emacs is unclear, the history is well-known from the middle of 1985. A time line is:

- GNU Emacs version 15 (15.34) was released sometime in 1984 or 1985 and shared some code with a version of Emacs written by James Gosling (the same James Gosling who later created the Java language).
- GNU Emacs version 16 (first released version was 16.56) was released on July 15, 1985. All Gosling code was removed due to potential copyright problems with the code.
- version 16.57: released on September 16, 1985.
- versions 16.58, 16.59: released on September 17, 1985.
- version 16.60: released on September 19, 1985. These later version 16's incorporated patches from the net, esp. for getting Emacs to work under System V.
- version 17.36 (first official v17 release) released on December 20, 1985. Included a TeX-able user manual. First official unpatched version that worked on vanilla System V machines.
- version 17.43 (second official v17 release) released on January 25, 1986.
- version 17.45 released on January 30, 1986.
- version 17.46 released on February 4, 1986.
- version 17.48 released on February 10, 1986.
- version 17.49 released on February 12, 1986.
- version 17.55 released on March 18, 1986.
- version 17.57 released on March 27, 1986.
- version 17.58 released on April 4, 1986.
- version 17.61 released on April 12, 1986.
- version 17.63 released on May 7, 1986.
- version 17.64 released on May 12, 1986.

- version 18.24 (a beta version) released on October 2, 1986.
- version 18.30 (a beta version) released on November 15, 1986.
- version 18.31 (a beta version) released on November 23, 1986.
- version 18.32 (a beta version) released on December 7, 1986.
- version 18.33 (a beta version) released on December 12, 1986.
- version 18.35 (a beta version) released on January 5, 1987.
- version 18.36 (a beta version) released on January 21, 1987.
- January 27, 1987: The Great Usenet Renaming. net.emacs is now comp.emacs.
- version 18.37 (a beta version) released on February 12, 1987.
- version 18.38 (a beta version) released on March 3, 1987.
- version 18.39 (a beta version) released on March 14, 1987.
- version 18.40 (a beta version) released on March 18, 1987.
- version 18.41 (the first "official" release) released on March 22, 1987.
- version 18.45 released on June 2, 1987.
- version 18.46 released on June 9, 1987.
- version 18.47 released on June 18, 1987.
- version 18.48 released on September 3, 1987.
- version 18.49 released on September 18, 1987.
- version 18.50 released on February 13, 1988.
- version 18.51 released on May 7, 1988.
- version 18.52 released on September 1, 1988.
- version 18.53 released on February 24, 1989.
- version 18.54 released on April 26, 1989.
- version 18.55 released on August 23, 1989. This is the earliest version that is still available by FTP.
- version 18.56 released on January 17, 1991.
- version 18.57 released late January, 1991.
- version 18.58 released ?????.
- version 18.59 released October 31, 1992.

1.2 Lucid Emacs

Lucid Emacs was developed by the (now-defunct) Lucid Inc., a maker of C++ and Lisp development environments. It began when Lucid decided they wanted to use Emacs as the editor and cornerstone of their C++ development environment (called "Energize"). They needed many features that were not available in the existing version of GNU Emacs (version 18.5something), in particular good and integrated support for GUI elements such as mouse support, multiple fonts, multiple window-system windows, etc. A branch of GNU Emacs called Epoch, written at the University of Illinois, existed that supplied many of these features; however, Lucid needed more than what existed in Epoch. At the time, the Free Software Foundation was working on version 19 of Emacs (this was sometime around 1991), which was planned to have similar features, and so Lucid decided to work with the Free Software Foundation. Their plan was to add features that they needed, and coordinate with the FSF so that the features would get included back into Emacs version 19.

Delays in the release of version 19 occurred, however (resulting in it finally being released more than a year after what was initially planned), and Lucid encountered unexpected technical resistance in getting their changes merged back into version 19, so they decided to release their own version of Emacs, which became Lucid Emacs 19.0.

The initial authors of Lucid Emacs were Matthieu Devin, Harlan Sexton, and Eric Benson, and the work was later taken over by Jamie Zawinski, who became "Mr. Lucid Emacs" for many releases.

A time line for Lucid Emacs/XEmacs is

- version 19.0 shipped with Energize 1.0, April 1992.
- version 19.1 released June 4, 1992.
- version 19.2 released June 19, 1992.
- version 19.3 released September 9, 1992.
- version 19.4 released January 21, 1993.
- version 19.5 was a repackaging of 19.4 with a few bug fixes and shipped with Energize 2.0. Never released to the net.
- version 19.6 released April 9, 1993.
- version 19.7 was a repackaging of 19.6 with a few bug fixes and shipped with Energize 2.1. Never released to the net.
- version 19.8 released September 6, 1993.
- version 19.9 released January 12, 1994.
- version 19.10 released May 27, 1994.
- version 19.11 (first XEmacs) released September 13, 1994.
- version 19.12 released June 23, 1995.
- version 19.13 released September 1, 1995.
- version 19.14 released June 23, 1996.
- version 20.0 released February 9, 1997.
- version 19.15 released March 28, 1997.
- version 20.1 (not released to the net) April 15, 1997.
- version 20.2 released May 16, 1997.
- version 19.16 released October 31, 1997.
- version 20.3 (the first stable version of XEmacs 20.x) released November 30, 1997. version 20.4 released February 28, 1998.

1.3 GNU Emacs 19

About a year after the initial release of Lucid Emacs, the FSF released a beta of their version of Emacs 19 (referred to here as "GNU Emacs"). By this time, the current version of Lucid Emacs was 19.6. (Strangely, the first released beta from the FSF was GNU Emacs 19.7.) A time line for GNU Emacs version 19 is

- version 19.8 (beta) released May 27, 1993.
- version 19.9 (beta) released May 27, 1993.
- version 19.10 (beta) released May 30, 1993.
- version 19.11 (beta) released June 1, 1993.
- version 19.12 (beta) released June 2, 1993.
- version 19.13 (beta) released June 8, 1993.
- version 19.14 (beta) released June 17, 1993.
- version 19.15 (beta) released June 19, 1993.
- version 19.16 (beta) released July 6, 1993.
- version 19.17 (beta) released late July, 1993.
- version 19.18 (beta) released August 9, 1993.
- version 19.19 (beta) released August 15, 1993.
- version 19.20 (beta) released November 17, 1993.
- version 19.21 (beta) released November 17, 1993.
- version 19.22 (beta) released November 28, 1993.
- version 19.23 (beta) released May 17, 1994.
- version 19.24 (beta) released May 16, 1994.
- version 19.25 (beta) released June 3, 1994.
- version 19.26 (beta) released September 11, 1994.
- version 19.27 (beta) released September 14, 1994.
- version 19.28 (first "official" release) released November 1, 1994.
- version 19.29 released June 21, 1995.
- version 19.30 released November 24, 1995.
- version 19.31 released May 25, 1996.
- version 19.32 released July 31, 1996.
- version 19.33 released August 11, 1996.
- version 19.34 released August 21, 1996.
- version 19.34b released September 6, 1996.

In some ways, GNU Emacs 19 was better than Lucid Emacs; in some ways, worse. Lucid soon began incorporating features from GNU Emacs 19 into Lucid Emacs; the work was mostly done by Richard Mlynarik, who had been working on and using GNU Emacs for a long time (back as far as version 16 or 17).

1.4 GNU Emacs 20

On February 2, 1997 work began on GNU Emacs to integrate Mule. The first release was made in September of that year.

A timeline for Emacs 20 is

- version 20.1 released September 17, 1997.
- version 20.2 released September 20, 1997.
- version 20.3 released August 19, 1998.

1.5 XEmacs

Around the time that Lucid was developing Energize, Sun Microsystems was developing their own development environment (called "SPARCWorks") and also decided to use Emacs. They joined forces with the Epoch team at the University of Illinois and later with Lucid. The maintainer of the last-released version of Epoch was Marc Andreessen, but he dropped out and the Epoch project, headed by Simon Kaplan, lured Chuck Thompson away from a system administration job to become the primary Lucid Emacs author for Epoch and Sun. Chuck's area of specialty became the redisplay engine (he replaced the old Lucid Emacs redisplay engine with a ported version from Epoch and then later rewrote it from scratch). Sun also hired Ben Wing (the author of Win-Emacs, a port of Lucid Emacs to Microsoft Windows 3.1) in 1993, for what was initially a one-month contract to fix some event problems but later became a many-year involvement, punctuated by a six-month contract with Amdahl Corporation.

In 1994, Sun and Lucid agreed to rename Lucid Emacs to XEmacs (a name not favorable to either company); the first release called XEmacs was version 19.11. In June 1994, Lucid folded and Jamie quit to work for the newly formed Mosaic Communications Corp., later Netscape Communications Corp. (co-founded by the same Marc Andreessen, who had quit his Epoch job to work on a graphical browser for the World Wide Web). Chuck then become the primary maintainer of XEmacs, and put out versions 19.11 through 19.14 in conjunction with Ben. For 19.12 and 19.13, Chuck added the new redisplay and many other display improvements and Ben added MULE support (support for Asian and other languages) and redesigned most of the internal Lisp subsystems to better support the MULE work and the various other features being added to XEmacs. After 19.14 Chuck retired as primary maintainer and Steve Baur stepped in.

Soon after 19.13 was released, work began in earnest on the MULE internationalization code and the source tree was divided into two development paths. The MULE version was initially called 19.20, but was soon renamed to 20.0. In 1996 Martin Buchholz of Sun Microsystems took over the care and feeding of it and worked on it in parallel with the 19.14 development that was occurring at the same time. After much work by Martin, it was decided to release 20.0 ahead of 19.15 in February 1997. The source tree remained divided until 20.2 when the version 19 source was finally retired at version 19.16.

In 1997, Sun finally dropped all pretense of support for XEmacs and Martin Buchholz left the company in November. Since then, and mostly for the previous year, because Steve Baur was never paid to work on XEmacs, XEmacs has existed solely on the contributions of volunteers from the Free Software Community. Starting from 1997, Hrvoje Niksic and Kyle Jones have figured prominently in XEmacs development.

Many attempts have been made to merge XEmacs and GNU Emacs, but they have consistently failed.

A more detailed history is contained in the XEmacs About page.

2 XEmacs From the Outside

XEmacs appears to the outside world as an editor, but it is really a Lisp environment. At its heart is a Lisp interpreter; it also "happens" to contain many specialized object types (e.g. buffers, windows, frames, events) that are useful for implementing an editor. Some of these objects (in particular windows and frames) have displayable representations, and XEmacs provides a function `redisplay()` that ensures that the display of all such objects matches their internal state. Most of the time, a standard Lisp environment is in a *read-eval-print* loop – i.e. "read some Lisp code, execute it, and print the results". XEmacs has a similar loop:

- read an event
- dispatch the event (i.e. "do it")
- redisplay

Reading an event is done using the Lisp function `next-event`, which waits for something to happen (typically, the user presses a key or moves the mouse) and returns an event object describing this. Dispatching an event is done using the Lisp function `dispatch-event`, which looks up the event in a keymap object (a particular kind of object that associates an event with a Lisp function) and calls that function. The function "does" what the user has requested by changing the state of particular frame objects, buffer objects, etc. Finally, `redisplay()` is called, which updates the display to reflect those changes just made. Thus is an "editor" born.

Note that you do not have to use XEmacs as an editor; you could just as well make it do your taxes, compute pi, play bridge, etc. You'd just have to write functions to do those operations in Lisp.

3 The Lisp Language

Lisp is a general-purpose language that is higher-level than C and in many ways more powerful than C. Powerful dialects of Lisp such as Common Lisp are probably much better languages for writing very large applications than is C. (Unfortunately, for many non-technical reasons C and its successor C++ have become the dominant languages for application development. These languages are both inadequate for extremely large applications, which is evidenced by the fact that newer, larger programs are becoming ever harder to write and are requiring ever more programmers despite great increases in C development environments; and by the fact that, although hardware speeds and reliability have been growing at an exponential rate, most software is still generally considered to be slow and buggy.)

The new Java language holds promise as a better general-purpose development language than C. Java has many features in common with Lisp that are not shared by C (this is not a coincidence, since Java was designed by James Gosling, a former Lisp hacker). This will be discussed more later.

For those used to C, here is a summary of the basic differences between C and Lisp:

1. Lisp has an extremely regular syntax. Every function, expression, and control statement is written in the form

   (*func arg1 arg2* ...)

   This is as opposed to C, which writes functions as

   `func(`*arg1*`, `*arg2*`, ...)`

   but writes expressions involving operators as (e.g.)

   arg1 `+` *arg2*

   and writes control statements as (e.g.)

   `while (`*expr*`) {` *statement1*`;` *statement2*`; ... }`

   Lisp equivalents of the latter two would be

   `(+` *arg1 arg2* ...)

   and

   `(while` *expr statement1 statement2* ...)

2. Lisp is a safe language. Assuming there are no bugs in the Lisp interpreter/compiler, it is impossible to write a program that "core dumps" or otherwise causes the machine to execute an illegal instruction. This is very different from C, where perhaps the most common outcome of a bug is exactly such a crash. A corollary of this is that the C operation of casting a pointer is impossible (and unnecessary) in Lisp, and that it is impossible to access memory outside the bounds of an array.
3. Programs and data are written in the same form. The parenthesis-enclosing form described above for statements is the same form used for the most common data type in Lisp, the list. Thus, it is possible to represent any Lisp program using Lisp data types, and for one program to construct Lisp statements and then dynamically *evaluate* them, or cause them to execute.
4. All objects are *dynamically typed.* This means that part of every object is an indication of what type it is. A Lisp program can manipulate an object without knowing what type

it is, and can query an object to determine its type. This means that, correspondingly, variables and function parameters can hold objects of any type and are not normally declared as being of any particular type. This is opposed to the *static typing* of C, where variables can hold exactly one type of object and must be declared as such, and objects do not contain an indication of their type because it's implicit in the variables they are stored in. It is possible in C to have a variable hold different types of objects (e.g. through the use of `void *` pointers or variable-argument functions), but the type information must then be passed explicitly in some other fashion, leading to additional program complexity.

5. Allocated memory is automatically reclaimed when it is no longer in use. This operation is called *garbage collection* and involves looking through all variables to see what memory is being pointed to, and reclaiming any memory that is not pointed to and is thus "inaccessible" and out of use. This is as opposed to C, in which allocated memory must be explicitly reclaimed using `free()`. If you simply drop all pointers to memory without freeing it, it becomes "leaked" memory that still takes up space. Over a long period of time, this can cause your program to grow and grow until it runs out of memory.
6. Lisp has built-in facilities for handling errors and exceptions. In C, when an error occurs, usually either the program exits entirely or the routine in which the error occurs returns a value indicating this. If an error occurs in a deeply-nested routine, then every routine currently called must unwind itself normally and return an error value back up to the next routine. This means that every routine must explicitly check for an error in all the routines it calls; if it does not do so, unexpected and often random behavior results. This is an extremely common source of bugs in C programs. An alternative would be to do a non-local exit using `longjmp()`, but that is often very dangerous because the routines that were exited past had no opportunity to clean up after themselves and may leave things in an inconsistent state, causing a crash shortly afterwards.

   Lisp provides mechanisms to make such non-local exits safe. When an error occurs, a routine simply signals that an error of a particular class has occurred, and a non-local exit takes place. Any routine can trap errors occurring in routines it calls by registering an error handler for some or all classes of errors. (If no handler is registered, a default handler, generally installed by the top-level event loop, is executed; this prints out the error and continues.) Routines can also specify cleanup code (called an *unwind-protect*) that will be called when control exits from a block of code, no matter how that exit occurs – i.e. even if a function deeply nested below it causes a non-local exit back to the top level.

   Note that this facility has appeared in some recent vintages of C, in particular Visual C++ and other PC compilers written for the Microsoft Win32 API.
7. In Emacs Lisp, local variables are *dynamically scoped*. This means that if you declare a local variable in a particular function, and then call another function, that subfunction can "see" the local variable you declared. This is actually considered a bug in Emacs Lisp and in all other early dialects of Lisp, and was corrected in Common Lisp. (In Common Lisp, you can still declare dynamically scoped variables if you want to – they are sometimes useful – but variables by default are *lexically scoped* as in C.)

For those familiar with Lisp, Emacs Lisp is modelled after MacLisp, an early dialect of Lisp developed at MIT (no relation to the Macintosh computer). There is a Common Lisp compatibility package available for Emacs that provides many of the features of Common Lisp.

The Java language is derived in many ways from C, and shares a similar syntax, but has the following features in common with Lisp (and different from C):

1. Java is a safe language, like Lisp.
2. Java provides garbage collection, like Lisp.
3. Java has built-in facilities for handling errors and exceptions, like Lisp.
4. Java has a type system that combines the best advantages of both static and dynamic typing. Objects (except very simple types) are explicitly marked with their type, as in dynamic typing; but there is a hierarchy of types and functions are declared to accept only certain types, thus providing the increased compile-time error-checking of static typing.

4 XEmacs From the Perspective of Building

The heart of XEmacs is the Lisp environment, which is written in C. This is contained in the '`src/`' subdirectory. Underneath '`src/`' are two subdirectories of header files: '`s/`' (header files for particular operating systems) and '`m/`' (header files for particular machine types). In practice the distinction between the two types of header files is blurred. These header files define or undefine certain preprocessor constants and macros to indicate particular characteristics of the associated machine or operating system. As part of the configure process, one '`s/`' file and one '`m/`' file is identified for the particular environment in which XEmacs is being built.

XEmacs also contains a great deal of Lisp code. This implements the operations that make XEmacs useful as an editor as well as just a Lisp environment, and also contains many add-on packages that allow XEmacs to browse directories, act as a mail and Usenet news reader, compile Lisp code, etc. There is actually more Lisp code than C code associated with XEmacs, but much of the Lisp code is peripheral to the actual operation of the editor. The Lisp code all lies in subdirectories underneath the '`lisp/`' directory.

The '`lwlib/`' directory contains C code that implements a generalized interface onto different X widget toolkits and also implements some widgets of its own that behave like Motif widgets but are faster, free, and in some cases more powerful. The code in this directory compiles into a library and is mostly independent from XEmacs.

The '`etc/`' directory contains various data files associated with XEmacs. Some of them are actually read by XEmacs at startup; others merely contain useful information of various sorts.

The '`lib-src/`' directory contains C code for various auxiliary programs that are used in connection with XEmacs. Some of them are used during the build process; others are used to perform certain functions that cannot conveniently be placed in the XEmacs executable (e.g. the '`movemail`' program for fetching mail out of '`/var/spool/mail`', which must be setgid to '`mail`' on many systems; and the '`gnuclient`' program, which allows an external script to communicate with a running XEmacs process).

The '`man/`' directory contains the sources for the XEmacs documentation. It is mostly in a form called Texinfo, which can be converted into either a printed document (by passing it through TeX) or into on-line documentation called *info files*.

The '`info/`' directory contains the results of formatting the XEmacs documentation as *info files*, for on-line use. These files are used when you enter the Info system using *`C-h i`* or through the Help menu.

The '`dynodump/`' directory contains auxiliary code used to build XEmacs on Solaris platforms.

The other directories contain various miscellaneous code and information that is not normally used or needed.

The first step of building involves running the '`configure`' program and passing it various parameters to specify any optional features you want and compiler arguments and such, as described in the '`INSTALL`' file. This determines what the build environment is, chooses the appropriate '`s/`' and '`m/`' file, and runs a series of tests to determine many details about your environment, such as which library functions are available and exactly how they work. (The '`s/`' and '`m/`' files only contain information that cannot be conveniently detected in this fashion.) The reason for running these tests is that it allows XEmacs to be compiled on a much wider variety of platforms than those that the XEmacs developers happen to be familiar with, including various sorts of hybrid platforms. This is especially important now that many operating systems give you a great deal of control over exactly what features you want installed, and allow for easy upgrading of parts of a system without upgrading the rest. It would be impossible to pre-determine and pre-specify the information for all possible configurations.

When configure is done running, it generates '`Makefile`'s and the file '`src/config.h`' (which describes the features of your system) from template files. You then run '`make`', which compiles the auxiliary code and programs in '`lib-src/`' and '`lwlib/`' and the main XEmacs executable in '`src/`'. The result of compiling and linking is an executable called '`temacs`', which is *not* the final XEmacs executable. '`temacs`' by itself is not intended to function as an editor or even display any windows on the screen, and if you simply run it, it will exit immediately. The '`Makefile`' runs '`temacs`' with certain options that cause it to initialize itself, read in a number of basic Lisp files, and then dump itself out into a new executable called '`xemacs`'. This new executable has been pre-initialized and contains pre-digested Lisp code that is necessary for the editor to function (this includes most basic Lisp functions, e.g. `not`, that can be defined in terms of other Lisp primitives; some initialization code that is called when certain objects, such as frames, are created; and all of the standard keybindings and code for the actions they result in). This executable, '`xemacs`', is the executable that you run to use the XEmacs editor.

Although '`temacs`' is not intended to be run as an editor, it can, by using the incantation `temacs -batch -l loadup.el run-temacs`. This is useful when the dumping procedure described above is broken, or when using certain program debugging tools such as Purify. These tools get mighty confused by the tricks played by the XEmacs build process, such as allocation memory in one process, and freeing it in the next.

5 XEmacs From the Inside

Internally, XEmacs is quite complex, and can be very confusing. To simplify things, it can be useful to think of XEmacs as containing an event loop that "drives" everything, and a number of other subsystems, such as a Lisp engine and a redisplay mechanism. Each of these other subsystems exists simultaneously in XEmacs, and each has a certain state. The flow of control continually passes in and out of these different subsystems in the course of normal operation of the editor.

It is important to keep in mind that, most of the time, the editor is "driven" by the event loop. Except during initialization and batch mode, all subsystems are entered directly or indirectly through the event loop, and ultimately, control exits out of all subsystems back up to the event loop. This cycle of entering a subsystem, exiting back out to the event loop, and starting another iteration of the event loop occurs once each keystroke, mouse motion, etc.

If you're trying to understand a particular subsystem (other than the event loop), think of it as a "daemon" process or "servant" that is responsible for one particular aspect of a larger system, and periodically receives commands or environment changes that cause it to do something. Ultimately, these commands and environment changes are always triggered by the event loop. For example:

- The window and frame mechanism is responsible for keeping track of what windows and frames exist, what buffers are in them, etc. It is periodically given commands (usually from the user) to make a change to the current window/frame state: i.e. create a new frame, delete a window, etc.
- The buffer mechanism is responsible for keeping track of what buffers exist and what text is in them. It is periodically given commands (usually from the user) to insert or delete text, create a buffer, etc. When it receives a text-change command, it notifies the redisplay mechanism.
- The redisplay mechanism is responsible for making sure that windows and frames are displayed correctly. It is periodically told (by the event loop) to actually "do its job", i.e. snoop around and see what the current state of the environment (mostly of the currently-existing windows, frames, and buffers) is, and make sure that that state matches what's actually displayed. It keeps lots and lots of information around (such as what is actually being displayed currently, and what the environment was last time it checked) so that it can minimize the work it has to do. It is also helped along in that whenever a relevant change to the environment occurs, the redisplay mechanism is told about this, so it has a pretty good idea of where it has to look to find possible changes and doesn't have to look everywhere.
- The Lisp engine is responsible for executing the Lisp code in which most user commands are written. It is entered through a call to `eval` or `funcall`, which occurs as a result of dispatching an event from the event loop. The functions it calls issue commands to the buffer mechanism, the window/frame subsystem, etc.
- The Lisp allocation subsystem is responsible for keeping track of Lisp objects. It is given commands from the Lisp engine to allocate objects, garbage collect, etc.

etc.

The important idea here is that there are a number of independent subsystems each with its own responsibility and persistent state, just like different employees in a company, and each subsystem is periodically given commands from other subsystems. Commands can flow from any one subsystem to any other, but there is usually some sort of hierarchy, with all commands originating from the event subsystem.

XEmacs is entered in `main()`, which is in '`emacs.c`'. When this is called the first time (in a properly-invoked '`temacs`'), it does the following:

1. It does some very basic environment initializations, such as determining where it and its directories (e.g. '`lisp/`' and '`etc/`') reside and setting up signal handlers.
2. It initializes the entire Lisp interpreter.
3. It sets the initial values of many built-in variables (including many variables that are visible to Lisp programs), such as the global keymap object and the built-in faces (a face is an object that describes the display characteristics of text). This involves creating Lisp objects and thus is dependent on step (2).
4. It performs various other initializations that are relevant to the particular environment it is running in, such as retrieving environment variables, determining the current date and the user who is running the program, examining its standard input, creating any necessary file descriptors, etc.
5. At this point, the C initialization is complete. A Lisp program that was specified on the command line (usually '`loadup.el`') is called (temacs is normally invoked as `temacs -batch -l loadup.el dump`). '`loadup.el`' loads all of the other Lisp files that are needed for the operation of the editor, calls the `dump-emacs` function to write out '`xemacs`', and then kills the temacs process.

When '`xemacs`' is then run, it only redoes steps (1) and (4) above; all variables already contain the values they were set to when the executable was dumped, and all memory that was allocated with `malloc()` is still around. (XEmacs knows whether it is being run as '`xemacs`' or '`temacs`' because it sets the global variable `initialized` to 1 after step (4) above.) At this point, '`xemacs`' calls a Lisp function to do any further initialization, which includes parsing the command-line (the C code can only do limited command-line parsing, which includes looking for the '`-batch`' and '`-l`' flags and a few other flags that it needs to know about before initialization is complete), creating the first frame (or *window* in standard window-system parlance), running the user's init file (usually the file '`.emacs`' in the user's home directory), etc. The function to do this is usually called `normal-top-level`; '`loadup.el`' tells the C code about this function by setting its name as the value of the Lisp variable `top-level`.

When the Lisp initialization code is done, the C code enters the event loop, and stays there for the duration of the XEmacs process. The code for the event loop is contained in '`keyboard.c`', and is called `Fcommand_loop_1()`. Note that this event loop could very well be written in Lisp, and in fact a Lisp version exists; but apparently, doing this makes XEmacs run noticeably slower.

Notice how much of the initialization is done in Lisp, not in C. In general, XEmacs tries to move as much code as is possible into Lisp. Code that remains in C is code that implements the Lisp interpreter itself, or code that needs to be very fast, or code that needs to do system calls or other such stuff that needs to be done in C, or code that needs to have access to "forbidden" structures. (One conscious aspect of the design of Lisp under

XEmacs is a clean separation between the external interface to a Lisp object's functionality and its internal implementation. Part of this design is that Lisp programs are forbidden from accessing the contents of the object other than through using a standard API. In this respect, XEmacs Lisp is similar to modern Lisp dialects but differs from GNU Emacs, which tends to expose the implementation and allow Lisp programs to look at it directly. The major advantage of hiding the implementation is that it allows the implementation to be redesigned without affecting any Lisp programs, including those that might want to be "clever" by looking directly at the object's contents and possibly manipulating them.)

Moving code into Lisp makes the code easier to debug and maintain and makes it much easier for people who are not XEmacs developers to customize XEmacs, because they can make a change with much less chance of obscure and unwanted interactions occurring than if they were to change the C code.

6 The XEmacs Object System (Abstractly Speaking)

At the heart of the Lisp interpreter is its management of objects. XEmacs Lisp contains many built-in objects, some of which are simple and others of which can be very complex; and some of which are very common, and others of which are rarely used or are only used internally. (Since the Lisp allocation system, with its automatic reclamation of unused storage, is so much more convenient than `malloc()` and `free()`, the C code makes extensive use of it in its internal operations.)

The basic Lisp objects are

`integer` 28 bits of precision, or 60 bits on 64-bit machines; the reason for this is described below when the internal Lisp object representation is described.

`float` Same precision as a double in C.

`cons` A simple container for two Lisp objects, used to implement lists and most other data structures in Lisp.

`char` An object representing a single character of text; chars behave like integers in many ways but are logically considered text rather than numbers and have a different read syntax. (the read syntax for a char contains the char itself or some textual encoding of it – for example, a Japanese Kanji character might be encoded as '`^[$(B#&^[(B`' using the ISO-2022 encoding standard – rather than the numerical representation of the char; this way, if the mapping between chars and integers changes, which is quite possible for Kanji characters and other extended characters, the same character will still be created. Note that some primitives confuse chars and integers. The worst culprit is `eq`, which makes a special exception and considers a char to be `eq` to its integer equivalent, even though in no other case are objects of two different types `eq`. The reason for this monstrosity is compatibility with existing code; the separation of char from integer came fairly recently.)

`symbol` An object that contains Lisp objects and is referred to by name; symbols are used to implement variables and named functions and to provide the equivalent of preprocessor constants in C.

`vector` A one-dimensional array of Lisp objects providing constant-time access to any of the objects; access to an arbitrary object in a vector is faster than for lists, but the operations that can be done on a vector are more limited.

`string` Self-explanatory; behaves much like a vector of chars but has a different read syntax and is stored and manipulated more compactly and efficiently.

`bit-vector`
A vector of bits; similar to a string in spirit.

`compiled-function`
An object describing compiled Lisp code, known as *byte code*.

`subr` An object describing a Lisp primitive.

Note that there is no basic "function" type, as in more powerful versions of Lisp (where it's called a *closure*). XEmacs Lisp does not provide the closure semantics implemented by Common Lisp and Scheme. The guts of a function in XEmacs Lisp are represented in one of four ways: a symbol specifying another function (when one function is an alias for another), a list containing the function's source code, a bytecode object, or a subr object. (In other words, given a symbol specifying the name of a function, calling `symbol-function` to retrieve the contents of the symbol's function cell will return one of these types of objects.)

XEmacs Lisp also contains numerous specialized objects used to implement the editor:

`buffer`
: Stores text like a string, but is optimized for insertion and deletion and has certain other properties that can be set.

`frame`
: An object with various properties whose displayable representation is a *window* in window-system parlance.

`window`
: A section of a frame that displays the contents of a buffer; often called a *pane* in window-system parlance.

`window-configuration`
: An object that represents a saved configuration of windows in a frame.

`device`
: An object representing a screen on which frames can be displayed; equivalent to a *display* in the X Window System and a *TTY* in character mode.

`face`
: An object specifying the appearance of text or graphics; it contains characteristics such as font, foreground color, and background color.

`marker`
: An object that refers to a particular position in a buffer and moves around as text is inserted and deleted to stay in the same relative position to the text around it.

`extent`
: Similar to a marker but covers a range of text in a buffer; can also specify properties of the text, such as a face in which the text is to be displayed, whether the text is invisible or unmodifiable, etc.

`event`
: Generated by calling `next-event` and contains information describing a particular event happening in the system, such as the user pressing a key or a process terminating.

`keymap`
: An object that maps from events (described using lists, vectors, and symbols rather than with an event object because the mapping is for classes of events, rather than individual events) to functions to execute or other events to recursively look up; the functions are described by name, using a symbol, or using lists to specify the function's code.

`glyph`
: An object that describes the appearance of an image (e.g. pixmap) on the screen; glyphs can be attached to the beginning or end of extents and in some future version of XEmacs will be able to be inserted directly into a buffer.

`process`
: An object that describes a connection to an externally-running process.

There are some other, less-commonly-encountered general objects:

`hashtable`
: An object that maps from an arbitrary Lisp object to another arbitrary Lisp object, using hashing for fast lookup.

`obarray` A limited form of hashtable that maps from strings to symbols; obarrays are used to look up a symbol given its name and are not actually their own object type but are kludgily represented using vectors with hidden fields (this representation derives from GNU Emacs).

`specifier`
: A complex object used to specify the value of a display property; a default value is given and different values can be specified for particular frames, buffers, windows, devices, or classes of device.

`char-table`
: An object that maps from chars or classes of chars to arbitrary Lisp objects; internally char tables use a complex nested-vector representation that is optimized to the way characters are represented as integers.

`range-table`
: An object that maps from ranges of integers to arbitrary Lisp objects.

And some strange special-purpose objects:

`charset`
`coding-system`
: Objects used when MULE, or multi-lingual/Asian-language, support is enabled.

`color-instance`
`font-instance`
`image-instance`
: An object that encapsulates a window-system resource; instances are mostly used internally but are exposed on the Lisp level for cleanness of the specifier model and because it's occasionally useful for Lisp program to create or query the properties of instances.

`subwindow`
: An object that encapsulate a *subwindow* resource, i.e. a window-system child window that is drawn into by an external process; this object should be integrated into the glyph system but isn't yet, and may change form when this is done.

`tooltalk-message`
`tooltalk-pattern`
: Objects that represent resources used in the ToolTalk interprocess communication protocol.

`toolbar-button`
: An object used in conjunction with the toolbar.

`x-resource`
: An object that encapsulates certain miscellaneous resources in the X window system, used only when Epoch support is enabled.

And objects that are only used internally:

opaque A generic object for encapsulating arbitrary memory; this allows you the generality of `malloc()` and the convenience of the Lisp object system.

lstream
: A buffering I/O stream, used to provide a unified interface to anything that can accept output or provide input, such as a file descriptor, a stdio stream, a chunk of memory, a Lisp buffer, a Lisp string, etc.; it's a Lisp object to make its memory management more convenient.

char-table-entry
: Subsidiary objects in the internal char-table representation.

extent-auxiliary
menubar-data
toolbar-data
: Various special-purpose objects that are basically just used to encapsulate memory for particular subsystems, similar to the more general "opaque" object.

symbol-value-forward
symbol-value-buffer-local
symbol-value-varalias
symbol-value-lisp-magic
: Special internal-only objects that are placed in the value cell of a symbol to indicate that there is something special with this variable – e.g. it has no value, it mirrors another variable, or it mirrors some C variable; there is really only one kind of object, called a *symbol-value-magic*, but it is sort-of halfway kludged into semi-different object types.

Some types of objects are *permanent*, meaning that once created, they do not disappear until explicitly destroyed, using a function such as `delete-buffer`, `delete-window`, `delete-frame`, etc. Others will disappear once they are not longer used, through the garbage collection mechanism. Buffers, frames, windows, devices, and processes are among the objects that are permanent. Note that some objects can go both ways: Faces can be created either way; extents are normally permanent, but detached extents (extents not referring to any text, as happens to some extents when the text they are referring to is deleted) are temporary. Note that some permanent objects, such as faces and coding systems, cannot be deleted. Note also that windows are unique in that they can be *undeleted* after having previously been deleted. (This happens as a result of restoring a window configuration.)

Note that many types of objects have a *read syntax*, i.e. a way of specifying an object of that type in Lisp code. When you load a Lisp file, or type in code to be evaluated, what really happens is that the function `read` is called, which reads some text and creates an object based on the syntax of that text; then `eval` is called, which possibly does something special; then this loop repeats until there's no more text to read. (`eval` only actually does something special with symbols, which causes the symbol's value to be returned, similar to referencing a variable; and with conses [i.e. lists], which cause a function invocation. All other values are returned unchanged.)

The read syntax

```
17297
```

converts to an integer whose value is 17297.

```
1.983e-4
```

converts to a float whose value is 1.983e-4, or .0001983.

```
?b
```

converts to a char that represents the lowercase letter b.

```
?^[$(B#&^[(B
```

(where '^[' actually is an 'ESC' character) converts to a particular Kanji character when using an ISO2022-based coding system for input. (To decode this gook: 'ESC' begins an escape sequence; 'ESC $ (' is a class of escape sequences meaning "switch to a 94x94 character set"; 'ESC $ (B' means "switch to Japanese Kanji"; '#' and '&' collectively index into a 94-by-94 array of characters [subtract 33 from the ASCII value of each character to get the corresponding index]; 'ESC (' is a class of escape sequences meaning "switch to a 94 character set"; 'ESC (B' means "switch to US ASCII". It is a coincidence that the letter 'B' is used to denote both Japanese Kanji and US ASCII. If the first 'B' were replaced with an 'A', you'd be requesting a Chinese Hanzi character from the GB2312 character set.)

```
"foobar"
```

converts to a string.

```
foobar
```

converts to a symbol whose name is `"foobar"`. This is done by looking up the string equivalent in the global variable `obarray`, whose contents should be an obarray. If no symbol is found, a new symbol with the name `"foobar"` is automatically created and added to `obarray`; this process is called *interning* the symbol.

```
(foo . bar)
```

converts to a cons cell containing the symbols `foo` and `bar`.

```
(1 a 2.5)
```

converts to a three-element list containing the specified objects (note that a list is actually a set of nested conses; see the XEmacs Lisp Reference).

```
[1 a 2.5]
```

converts to a three-element vector containing the specified objects.

```
#[... ... ... ...]
```

converts to a compiled-function object (the actual contents are not shown since they are not relevant here; look at a file that ends with '.elc' for examples).

```
#*01110110
```

converts to a bit-vector.

```
#s(range-table ... ...)
```

converts to a range table (the actual contents are not shown).

```
#s(char-table ... ...)
```

converts to a char table (the actual contents are not shown). (Note that the #s syntax is the general syntax for structures, which are not really implemented in XEmacs Lisp but should be.)

When an object is printed out (using `print` or a related function), the read syntax is used, so that the same object can be read in again.

The other objects do not have read syntaxes, usually because it does not really make sense to create them in this fashion (i.e. processes, where it doesn't make sense to have a subprocess created as a side effect of reading some Lisp code), or because they can't be created at all (e.g. subrs). Permanent objects, as a rule, do not have a read syntax; nor do most complex objects, which contain too much state to be easily initialized through a read syntax.

7 How Lisp Objects Are Represented in C

Lisp objects are represented in C using a 32- or 64-bit machine word (depending on the processor; i.e. DEC Alphas use 64-bit Lisp objects and most other processors use 32-bit Lisp objects). The representation stuffs a pointer together with a tag, as follows:

```
[ 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 ]
[ 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 ]

  ^ <---> <------------------------------------------------------>
  |  tag         a pointer to a structure, or an integer
  |
  `---> mark bit
```

The tag describes the type of the Lisp object. For integers and chars, the lower 28 bits contain the value of the integer or char; for all others, the lower 28 bits contain a pointer. The mark bit is used during garbage-collection, and is always 0 when garbage collection is not happening. Many macros that extract out parts of a Lisp object expect that the mark bit is 0, and will produce incorrect results if it's not. (The way that garbage collection works, basically, is that it loops over all places where Lisp objects could exist – this includes all global variables in C that contain Lisp objects [including `Vobarray`, the C equivalent of `obarray`; through this, all Lisp variables will get marked], plus various other places – and recursively scans through the Lisp objects, marking each object it finds by setting the mark bit. Then it goes through the lists of all objects allocated, freeing the ones that are not marked and turning off the mark bit of the ones that are marked.)

Lisp objects use the typedef `Lisp_Object`, but the actual C type used for the Lisp object can vary. It can be either a simple type (`long` on the DEC Alpha, `int` on other machines) or a structure whose fields are bit fields that line up properly (actually, a union of structures that's used). Generally the simple integral type is preferable because it ensures that the compiler will actually use a machine word to represent the object (some compilers will use more general and less efficient code for unions and structs even if they can fit in a machine word). The union type, however, has the advantage of stricter type checking (if you accidentally pass an integer where a Lisp object is desired, you get a compile error), and it makes it easier to decode Lisp objects when debugging. The choice of which type to use is determined by the presence or absence of the preprocessor constant `USE_UNION_TYPE`.

Note that there are only eight types that the tag can represent, but many more actual types than this. This is handled by having one of the tag types specify a meta-type called a *record*; for all such objects, the first four bytes of the pointed-to structure indicate what the actual type is.

Note also that having 28 bits for pointers and integers restricts a lot of things to 256 megabytes of memory. (Basically, enough pointers and indices and whatnot get stuffed into Lisp objects that the total amount of memory used by XEmacs can't grow above 256 megabytes. In older versions of XEmacs and GNU Emacs, the tag was 5 bits wide, allowing for 32 types, which was more than the actual number of types that existed at the time, and no "record" type was necessary. However, this limited the editor to 64 megabytes total, which some users who edited large files might conceivably exceed.)

Also, note that there is an implicit assumption here that all pointers are low enough that the top bits are all zero and can just be chopped off. On standard machines that allocate memory from the bottom up (and give each process its own address space), this works fine. Some machines, however, put the data space somewhere else in memory (e.g. beginning at 0x80000000). Those machines cope by defining `DATA_SEG_BITS` in the corresponding '`m/`' or '`s/`' file to the proper mask. Then, pointers retrieved from Lisp objects are automatically OR'ed with this value prior to being used.

A corollary of the previous paragraph is that **(pointers to) stack-allocated structures cannot be put into Lisp objects**. The stack is generally located near the top of memory; if you put such a pointer into a Lisp object, it will get its top bits chopped off, and you will lose.

Various macros are used to construct Lisp objects and extract the components. Macros of the form `XINT()`, `XCHAR()`, `XSTRING()`, `XSYMBOL()`, etc. mask out the pointer/integer field and cast it to the appropriate type. All of the macros that construct pointers will `OR` with `DATA_SEG_BITS` if necessary. `XINT()` needs to be a bit tricky so that negative numbers are properly sign-extended: Usually it does this by shifting the number four bits to the left and then four bits to the right. This assumes that the right-shift operator does an arithmetic shift (i.e. it leaves the most-significant bit as-is rather than shifting in a zero, so that it mimics a divide-by-two even for negative numbers). Not all machines/compilers do this, and on the ones that don't, a more complicated definition is selected by defining `EXPLICIT_SIGN_EXTEND`.

Note that when `ERROR_CHECK_TYPECHECK` is defined, the extractor macros become more complicated – they check the tag bits and/or the type field in the first four bytes of a record type to ensure that the object is really of the correct type. This is great for catching places where an incorrect type is being dereferenced – this typically results in a pointer being dereferenced as the wrong type of structure, with unpredictable (and sometimes not easily traceable) results.

There are similar `XSET`*TYPE*`()` macros that construct a Lisp object. These macros are of the form `XSET`*TYPE* `(`*lvalue*`, `*result*`)`, i.e. they have to be a statement rather than just used in an expression. The reason for this is that standard C doesn't let you "construct" a structure (but GCC does). Granted, this sometimes isn't too convenient; for the case of integers, at least, you can use the function `make_int()`, which constructs and *returns* an integer Lisp object. Note that the `XSET`*TYPE*`()` macros are also affected by `ERROR_CHECK_TYPECHECK` and make sure that the structure is of the right type in the case of record types, where the type is contained in the structure.

8 Rules When Writing New C Code

The XEmacs C Code is extremely complex and intricate, and there are many rules that are more or less consistently followed throughout the code. Many of these rules are not obvious, so they are explained here. It is of the utmost importance that you follow them. If you don't, you may get something that appears to work, but which will crash in odd situations, often in code far away from where the actual breakage is.

8.1 General Coding Rules

Almost every module contains a `syms_of_*()` function and a `vars_of_*()` function. The former declares any Lisp primitives you have defined and defines any symbols you will be using. The latter declares any global Lisp variables you have added and initializes global C variables in the module. For each such function, declare it in '`symsinit.h`' and make sure it's called in the appropriate place in '`emacs.c`'. **Important**: There are stringent requirements on exactly what can go into these functions. See the comment in '`emacs.c`'. The reason for this is to avoid obscure unwanted interactions during initialization. If you don't follow these rules, you'll be sorry! If you want to do anything that isn't allowed, create a `complex_vars_of_*()` function for it. Doing this is tricky, though: You have to make sure your function is called at the right time so that all the initialization dependencies work out.

Every module includes '`<config.h>`' (angle brackets so that '`--srcdir`' works correctly; '`config.h`' may or may not be in the same directory as the C sources) and '`lisp.h`'. '`config.h`' should always be included before any other header files (including system header files) to ensure that certain tricks played by various '`s/`' and '`m/`' files work out correctly.

All global and static variables that are to be modifiable must be declared uninitialized. This means that you may not use the "declare with initializer" form for these variables, such as `int some_variable = 0;`. The reason for this has to do with some kludges done during the dumping process: If possible, the initialized data segment is re-mapped so that it becomes part of the (unmodifiable) code segment in the dumped executable. This allows this memory to be shared among multiple running XEmacs processes. XEmacs is careful to place as much constant data as possible into initialized variables (in particular, into what's called the *pure space* – see below) during the '`temacs`' phase.

Please note: This kludge only works on a few systems nowadays, and is rapidly becoming irrelevant because most modern operating systems provide *copy-on-write* semantics. All data is initially shared between processes, and a private copy is automatically made (on a page-by-page basis) when a process first attempts to write to a page of memory.

Formerly, there was a requirement that static variables not be declared inside of functions. This had to do with another hack along the same vein as what was just described: old USG systems put statically-declared variables in the initialized data space, so those header files had a `#define static` declaration. (That way, the data-segment remapping described above could still work.) This fails badly on static variables inside of functions, which suddenly become automatic variables; therefore, you weren't supposed to have any of them. This awful kludge has been removed in XEmacs because

1. almost all of the systems that used this kludge ended up having to disable the data-segment remapping anyway;

2. the only systems that didn't were extremely outdated ones;
3. this hack completely messed up inline functions.

8.2 Writing Lisp Primitives

Lisp primitives are Lisp functions implemented in C. The details of interfacing the C function so that Lisp can call it are handled by a few C macros. The only way to really understand how to write new C code is to read the source, but we can explain some things here.

An example of a special form is the definition of `or`, from '`eval.c`'. (An ordinary function would have the same general appearance.)

```
DEFUN ("or", For, 0, UNEVALLED, 0, /*
Eval args until one of them yields non-nil, then return that value.
The remaining args are not evalled at all.
If all args return nil, return nil.
*/
       (args))
{
  /* This function can GC */
  Lisp_Object val = Qnil;
  struct gcpro gcpro1;

  GCPRO1 (args);

  while (!NILP (args))
    {
      val = Feval (XCAR (args));
      if (!NILP (val))
        break;
      args = XCDR (args);
    }

  UNGCPRO;
  return val;
}
```

Let's start with a precise explanation of the arguments to the `DEFUN` macro. Here is a template for them:

```
DEFUN (lname, fname, min, max, interactive, /*
docstring
*/
     (arglist) )
```

lname This string is the name of the Lisp symbol to define as the function name; in the example above, it is `"or"`.

fname This is the C function name for this function. This is the name that is used in C code for calling the function. The name is, by convention, '`F`' prepended to the Lisp name, with all dashes ('`-`') in the Lisp name changed to underscores. Thus, to call this function from C code, call `For`. Remember that the arguments are of type `Lisp_Object`; various macros and functions for creating values of type `Lisp_Object` are declared in the file '`lisp.h`'.

Primitives whose names are special characters (e.g. `+` or `<`) are named by spelling out, in some fashion, the special character: e.g. `Fplus()` or `Flss()`. Primitives whose names begin with normal alphanumeric characters but also contain special characters are spelled out in some creative way, e.g. `let*` becomes `FletX()`.

Each function also has an associated structure that holds the data for the subr object that represents the function in Lisp. This structure conveys the Lisp symbol name to the initialization routine that will create the symbol and store the subr object as its definition. The C variable name of this structure is always '`S`' prepended to the *fname*. You hardly ever need to be aware of the existence of this structure.

min This is the minimum number of arguments that the function requires. The function `or` allows a minimum of zero arguments.

max This is the maximum number of arguments that the function accepts, if there is a fixed maximum. Alternatively, it can be `UNEVALLED`, indicating a special form that receives unevaluated arguments, or `MANY`, indicating an unlimited number of evaluated arguments (the equivalent of `&rest`). Both `UNEVALLED` and `MANY` are macros. If *max* is a number, it may not be less than *min* and it may not be greater than 8. (If you need to add a function with more than 8 arguments, either use the `MANY` form or edit the definition of `DEFUN` in '`lisp.h`'. If you do the latter, make sure to also add another clause to the switch statement in `primitive_funcall()`.)

interactive

This is an interactive specification, a string such as might be used as the argument of `interactive` in a Lisp function. In the case of `or`, it is 0 (a null pointer), indicating that `or` cannot be called interactively. A value of `""` indicates a function that should receive no arguments when called interactively.

docstring This is the documentation string. It is written just like a documentation string for a function defined in Lisp; in particular, the first line should be a single sentence. Note how the documentation string is enclosed in a comment, none of the documentation is placed on the same lines as the comment-start and comment-end characters, and the comment-start characters are on the same line as the interactive specification. '`make-docfile`', which scans the C files for documentation strings, is very particular about what it looks for, and will not properly extract the doc string if it's not in this exact format.

You are free to put the various arguments to `DEFUN` on separate lines to avoid overly long lines. However, make sure to put the comment-start characters for the doc string on the same line as the interactive specification, and put a newline directly after them (and before the comment-end characters).

arglist This is the comma-separated list of arguments to the C function. For a function with a fixed maximum number of arguments, provide a C argument for each Lisp argument. In this case, unlike regular C functions, the types of the arguments are not declared; they are simply always of type `Lisp_Object`.

The names of the C arguments will be used as the names of the arguments to the Lisp primitive as displayed in its documentation, modulo the same concerns described above for `F...` names (in particular, underscores in the C arguments become dashes in the Lisp arguments).

There is one additional kludge: A trailing '_' on the C argument is discarded when forming the Lisp argument. This allows C language reserved words (like `default`) or global symbols (like `dirname`) to be used as argument names without compiler warnings or errors.

A Lisp function with *max* = `UNEVALLED` is a *special form*; its arguments are not evaluated. Instead it receives one argument of type `Lisp_Object`, a (Lisp) list of the unevaluated arguments, conventionally named `(args)`.

When a Lisp function has no upper limit on the number of arguments, specify *max* = `MANY`. In this case its implementation in C actually receives exactly two arguments: the number of Lisp arguments (an `int`) and the address of a block containing their values (a `Lisp_Object *`). In this case only are the C types specified in the *arglist*: `(int nargs, Lisp_Object *args)`.

Within the function `For` itself, note the use of the macros `GCPRO1` and `UNGCPRO`. `GCPRO1` is used to "protect" a variable from garbage collection—to inform the garbage collector that it must look in that variable and regard its contents as an accessible object. This is necessary whenever you call `Feval` or anything that can directly or indirectly call `Feval` (this includes the `QUIT` macro!). At such a time, any Lisp object that you intend to refer to again must be protected somehow. `UNGCPRO` cancels the protection of the variables that are protected in the current function. It is necessary to do this explicitly.

The macro `GCPRO1` protects just one local variable. If you want to protect two, use `GCPRO2` instead; repeating `GCPRO1` will not work. Macros `GCPRO3` and `GCPRO4` also exist.

These macros implicitly use local variables such as `gcpro1`; you must declare these explicitly, with type `struct gcpro`. Thus, if you use `GCPRO2`, you must declare `gcpro1` and `gcpro2`.

Note also that the general rule is *caller-protects*; i.e. you are only responsible for protecting those Lisp objects that you create. Any objects passed to you as parameters should have been protected by whoever created them, so you don't in general have to protect them. `For` is an exception; it protects its parameters to provide extra assurance against Lisp primitives elsewhere that are incorrectly written, and against malicious self-modifying code. There are a few other standard functions that also do this.

`GCPRO`ing is perhaps the trickiest and most error-prone part of XEmacs coding. It is **extremely** important that you get this right and use a great deal of discipline when writing this code. See Section 10.3 [`GCPRO`ing], page 63, for full details on how to do this.

What `DEFUN` actually does is declare a global structure of type `Lisp_Subr` whose name begins with capital '`SF`' and which contains information about the primitive (e.g. a pointer to the function, its minimum and maximum allowed arguments, a string describing its Lisp name); `DEFUN` then begins a normal C function declaration using the `F...` name. The Lisp subr object that is the function definition of a primitive (i.e. the object in the function slot of the symbol that names the primitive) actually points to this 'SF' structure; when `Feval` encounters a subr, it looks in the structure to find out how to call the C function.

Defining the C function is not enough to make a Lisp primitive available; you must also create the Lisp symbol for the primitive (the symbol is *interned*; see Section 13.2 [Obarrays], page 85) and store a suitable subr object in its function cell. (If you don't do this, the primitive won't be seen by Lisp code.) The code looks like this:

```
DEFSUBR (fname);
```

Here *fname* is the name you used as the second argument to `DEFUN`.

This call to `DEFSUBR` should go in the `syms_of_*()` function at the end of the module. If no such function exists, create it and make sure to also declare it in '`symsinit.h`' and call it from the appropriate spot in `main()`. See Section 8.1 [General Coding Rules], page 27.

Note that C code cannot call functions by name unless they are defined in C. The way to call a function written in Lisp from C is to use `Ffuncall`, which embodies the Lisp function `funcall`. Since the Lisp function `funcall` accepts an unlimited number of arguments, in C it takes two: the number of Lisp-level arguments, and a one-dimensional array containing their values. The first Lisp-level argument is the Lisp function to call, and the rest are the arguments to pass to it. Since `Ffuncall` can call the evaluator, you must protect pointers from garbage collection around the call to `Ffuncall`. (However, `Ffuncall` explicitly protects all of its parameters, so you don't have to protect any pointers passed as parameters to it.)

The C functions `call0`, `call1`, `call2`, and so on, provide handy ways to call a Lisp function conveniently with a fixed number of arguments. They work by calling `Ffuncall`.

'`eval.c`' is a very good file to look through for examples; '`lisp.h`' contains the definitions for some important macros and functions.

8.3 Adding Global Lisp Variables

Global variables whose names begin with '`Q`' are constants whose value is a symbol of a particular name. The name of the variable should be derived from the name of the symbol using the same rules as for Lisp primitives. These variables are initialized using a call to `defsymbol()` in the `syms_of_*()` function. (This call interns a symbol, sets the C variable to the resulting Lisp object, and calls `staticpro()` on the C variable to tell the garbage-collection mechanism about this variable. What `staticpro()` does is add a pointer to the variable to a large global array; when garbage-collection happens, all pointers listed in the array are used as starting points for marking Lisp objects. This is important because it's quite possible that the only current reference to the object is the C variable. In the case of symbols, the `staticpro()` doesn't matter all that much because the symbol is contained in `obarray`, which is itself `staticpro()`ed. However, it's possible that a naughty user could do something like uninterning the symbol out of `obarray` or even setting `obarray` to a different value [although this is likely to make XEmacs crash!].)

Please note: It is potentially deadly if you declare a '`Q...`' variable in two different modules. The two calls to `defsymbol()` are no problem, but some linkers will complain about multiply-defined symbols. The most insidious aspect of this is that often the link will succeed anyway, but then the resulting executable will sometimes crash in obscure ways during certain operations! To avoid this problem, declare any symbols with common names (such as `text`) that are not obviously associated with this particular module in the module '`general.c`'.

Global variables whose names begin with '`V`' are variables that contain Lisp objects. The convention here is that all global variables of type `Lisp_Object` begin with '`V`', and all others don't (including integer and boolean variables that have Lisp equivalents). Most of the time, these variables have equivalents in Lisp, but some don't. Those that do are declared this way by a call to `DEFVAR_LISP()` in the `vars_of_*()` initializer for the module. What this does is create a special *symbol-value-forward* Lisp object that contains a pointer to the C variable, intern a symbol whose name is as specified in the call to `DEFVAR_LISP()`, and set its value to the symbol-value-forward Lisp object; it also calls `staticpro()` on the C variable to tell the garbage-collection mechanism about the variable. When `eval` (or actually `symbol-value`) encounters this special object in the process of retrieving a variable's value, it follows the indirection to the C variable and gets its value. `setq` does similar things so that the C variable gets changed.

Whether or not you `DEFVAR_LISP()` a variable, you need to initialize it in the `vars_of_*()` function; otherwise it will end up as all zeroes, which is the integer 0 (*not* `nil`), and this is probably not what you want. Also, if the variable is not `DEFVAR_LISP()`ed, **you must call** `staticpro()` on the C variable in the `vars_of_*()` function. Otherwise, the garbage-collection mechanism won't know that the object in this variable is in use, and will happily collect it and reuse its storage for another Lisp object, and you will be the one who's unhappy when you can't figure out how your variable got overwritten.

8.4 Coding for Mule

Although Mule support is not compiled by default in XEmacs, many people are using it, and we consider it crucial that new code works correctly with multibyte characters. This is not hard; it is only a matter of following several simple user-interface guidelines. Even if you never compile with Mule, with a little practice you will find it quite easy to code Mule-correctly.

Note that these guidelines are not necessarily tied to the current Mule implementation; they are also a good idea to follow on the grounds of code generalization for future I18N work.

8.4.1 Character-Related Data Types

First, we will list the basic character-related datatypes used by XEmacs. Note that the separate `typedef`s are not required for the code to work (all of them boil down to `unsigned char` or `int`), but they improve clarity of code a great deal, because one glance at the declaration can tell the intended use of the variable.

`Emchar` An `Emchar` holds a single Emacs character.

Obviously, the equality between characters and bytes is lost in the Mule world. Characters can be represented by one or more bytes in the buffer, and `Emchar` is the C type large enough to hold any character.

Without Mule support, an `Emchar` is equivalent to an `unsigned char`.

`Bufbyte` The data representing the text in a buffer or string is logically a set of `Bufbyte`s.

XEmacs does not work with character formats all the time; when reading characters from the outside, it decodes them to an internal format, and likewise encodes them when writing. `Bufbyte` (in fact `unsigned char`) is the basic unit of XEmacs internal buffers and strings format.

One character can correspond to one or more `Bufbyte`s. In the current implementation, an ASCII character is represented by the same `Bufbyte`, and extended characters are represented by a sequence of `Bufbyte`s.

Without Mule support, a `Bufbyte` is equivalent to an `Emchar`.

`Bufpos`
`Charcount`
: A `Bufpos` represents a character position in a buffer or string. A `Charcount` represents a number (count) of characters. Logically, subtracting two `Bufpos` values yields a `Charcount` value. Although all of these are `typedef`ed to `int`, we use them in preference to `int` to make it clear what sort of position is being used.

  `Bufpos` and `Charcount` values are the only ones that are ever visible to Lisp.

`Bytind`
`Bytecount`
: A `Bytind` represents a byte position in a buffer or string. A `Bytecount` represents the distance between two positions in bytes. The relationship between `Bytind` and `Bytecount` is the same as the relationship between `Bufpos` and `Charcount`.

`Extbyte`
`Extcount`
: When dealing with the outside world, XEmacs works with `Extbyte`s, which are equivalent to `unsigned char`. Obviously, an `Extcount` is the distance between two `Extbyte`s. Extbytes and Extcounts are not all that frequent in XEmacs code.

8.4.2 Working With Character and Byte Positions

Now that we have defined the basic character-related types, we can look at the macros and functions designed for work with them and for conversion between them. Most of these macros are defined in '`buffer.h`', and we don't discuss all of them here, but only the most important ones. Examining the existing code is the best way to learn about them.

`MAX_EMCHAR_LEN`
: This preprocessor constant is the maximum number of buffer bytes per Emacs character, i.e. the byte length of an `Emchar`. It is useful when allocating temporary strings to keep a known number of characters. For instance:

```
{
  Charcount cclen;
  ...
  {
    /* Allocate place for cclen characters. */
    Bufbyte *tmp_buf = (Bufbyte *)alloca (cclen * MAX_EMCHAR_LEN);
...
```

If you followed the previous section, you can guess that, logically, multiplying a `Charcount` value with `MAX_EMCHAR_LEN` produces a `Bytecount` value.

In the current Mule implementation, `MAX_EMCHAR_LEN` equals 4. Without Mule, it is 1.

`charptr_emchar`

`set_charptr_emchar`

`charptr_emchar` macro takes a `Bufbyte` pointer and returns the underlying `Emchar`. If it were a function, its prototype would be:

```
Emchar charptr_emchar (Bufbyte *p);
```

`set_charptr_emchar` stores an `Emchar` to the specified byte position. It returns the number of bytes stored:

```
Bytecount set_charptr_emchar (Bufbyte *p, Emchar c);
```

It is important to note that `set_charptr_emchar` is safe only for appending a character at the end of a buffer, not for overwriting a character in the middle. This is because the width of characters varies, and `set_charptr_emchar` cannot resize the string if it writes, say, a two-byte character where a single-byte character used to reside.

A typical use of `set_charptr_emchar` can be demonstrated by this example, which copies characters from buffer *buf* to a temporary string of Bufbytes.

```
{
  Bufpos pos;
  for (pos = beg; pos < end; pos++)
    {
      Emchar c = BUF_FETCH_CHAR (buf, pos);
      p += set_charptr_emchar (buf, c);
    }
}
```

Note how `set_charptr_emchar` is used to store the `Emchar` and increment the counter, at the same time.

`INC_CHARPTR`

`DEC_CHARPTR`

These two macros increment and decrement a `Bufbyte` pointer, respectively. The pointer needs to be correctly positioned at the beginning of a valid character position.

Without Mule support, `INC_CHARPTR (p)` and `DEC_CHARPTR (p)` simply expand to `p++` and `p--`, respectively.

`bytecount_to_charcount`

Given a pointer to a text string and a length in bytes, return the equivalent length in characters.

```
Charcount bytecount_to_charcount (Bufbyte *p, Bytecount bc);
```

`charcount_to_bytecount`

Given a pointer to a text string and a length in characters, return the equivalent length in bytes.

```
Bytecount charcount_to_bytecount (Bufbyte *p, Charcount cc);
```

`charptr_n_addr`
: Return a pointer to the beginning of the character offset *cc* (in characters) from *p*.

```
Bufbyte *charptr_n_addr (Bufbyte *p, Charcount cc);
```

8.4.3 Conversion of External Data

When an external function, such as a C library function, returns a `char` pointer, you should never treat it as `Bufbyte`. This is because these returned strings may contain 8bit characters which can be misinterpreted by XEmacs, and cause a crash. Instead, you should use a conversion macro. Many different conversion macros are defined in '`buffer.h`', so I will try to order them logically, by direction and by format.

Thus the basic conversion macros are `GET_CHARPTR_INT_DATA_ALLOCA` and `GET_CHARPTR_EXT_DATA_ALLOCA`. The former is used to convert external data to internal format, and the latter is used to convert the other way around. The arguments each of these receives are *ptr* (pointer to the text in external format), *len* (length of texts in bytes), *fmt* (format of the external text), *ptr_out* (lvalue to which new text should be copied), and *len_out* (lvalue which will be assigned the length of the internal text in bytes). The resulting text is stored to a stack-allocated buffer. If the text doesn't need changing, these macros will do nothing, except for setting *len_out*.

Currently meaningful formats are `FORMAT_BINARY`, `FORMAT_FILENAME`, `FORMAT_OS`, and `FORMAT_CTEXT`.

The two macros above take many arguments which makes them unwieldy. For this reason, several convenience macros are defined with obvious functionality, but accepting less arguments:

`GET_C_CHARPTR_EXT_DATA_ALLOCA`
`GET_C_CHARPTR_INT_DATA_ALLOCA`
: These two macros work on "C char pointers", which are zero-terminated, and thus do not need *len* or *len_out* parameters.

`GET_STRING_EXT_DATA_ALLOCA`
`GET_C_STRING_EXT_DATA_ALLOCA`
: These two macros work on Lisp strings, thus also not needing a *len* parameter. However, `GET_STRING_EXT_DATA_ALLOCA` still provides a *len_out* parameter. Note that for Lisp strings only one conversion direction makes sense.

`GET_C_CHARPTR_EXT_BINARY_DATA_ALLOCA`
`GET_C_CHARPTR_EXT_FILENAME_DATA_ALLOCA`
`GET_C_CHARPTR_EXT_CTEXT_DATA_ALLOCA`
`...`
: These macros are a combination of the above, but with the *fmt* argument encoded into the name of the macro.

8.4.4 General Guidelines for Writing Mule-Aware Code

This section contains some general guidance on how to write Mule-aware code, as well as some pitfalls you should avoid.

Never use `char` *and* `char *`.
In XEmacs, the use of `char` and `char *` is almost always a mistake. If you want to manipulate an Emacs character from "C", use `Emchar`. If you want to examine a specific octet in the internal format, use `Bufbyte`. If you want a Lisp-visible character, use a `Lisp_Object` and `make_char`. If you want a pointer to move through the internal text, use `Bufbyte *`. Also note that you almost certainly do not need `Emchar *`.

Be careful not to confuse `Charcount`, `Bytecount`, *and* `Bufpos`.
The whole point of using different types is to avoid confusion about the use of certain variables. Lest this effect be nullified, you need to be careful about using the right types.

Always convert external data
It is extremely important to always convert external data, because XEmacs can crash if unexpected 8bit sequences are copied to its internal buffers literally.

This means that when a system function, such as `readdir`, returns a string, you need to convert it using one of the conversion macros described in the previous chapter, before passing it further to Lisp. In the case of `readdir`, you would use the `GET_C_CHARPTR_INT_FILENAME_DATA_ALLOCA` macro.

Also note that many internal functions, such as `make_string`, accept Bufbytes, which removes the need for them to convert the data they receive. This increases efficiency because that way external data needs to be decoded only once, when it is read. After that, it is passed around in internal format.

8.4.5 An Example of Mule-Aware Code

As an example of Mule-aware code, we shall will analyze the `string` function, which conses up a Lisp string from the character arguments it receives. Here is the definition, pasted from `alloc.c`:

```
DEFUN ("string", Fstring, 0, MANY, 0, /*
Concatenate all the argument characters and make the result a string.
*/
       (int nargs, Lisp_Object *args))
{
  Bufbyte *storage = alloca_array (Bufbyte, nargs * MAX_EMCHAR_LEN);
  Bufbyte *p = storage;

  for (; nargs; nargs--, args++)
    {
      Lisp_Object lisp_char = *args;
      CHECK_CHAR_COERCE_INT (lisp_char);
      p += set_charptr_emchar (p, XCHAR (lisp_char));
    }
  return make_string (storage, p - storage);
}
```

Now we can analyze the source line by line.

Obviously, string will be as long as there are arguments to the function. This is why we allocate `MAX_EMCHAR_LEN` * *nargs* bytes on the stack, i.e. the worst-case number of bytes for *nargs* `Emchar`s to fit in the string.

Then, the loop checks that each element is a character, converting integers in the process. Like many other functions in XEmacs, this function silently accepts integers where characters are expected, for historical and compatibility reasons. Unless you know what you are doing, `CHECK_CHAR` will also suffice. `XCHAR (lisp_char)` extracts the `Emchar` from the `Lisp_Object`, and `set_charptr_emchar` stores it to storage, increasing `p` in the process.

Other instructing examples of correct coding under Mule can be found all over XEmacs code. For starters, I recommend `Fnormalize_menu_item_name` in '`menubar.c`'. After you have understood this section of the manual and studied the examples, you can proceed writing new Mule-aware code.

8.5 Techniques for XEmacs Developers

To make a quantified XEmacs, do: `make quantmacs`.

You simply can't dump Quantified and Purified images. Run the image like so: `quantmacs -batch -l loadup.el run-temacs -q`.

Before you go through the trouble, are you compiling with all debugging and error-checking off? If not try that first. Be warned that while Quantify is directly responsible for quite a few optimizations which have been made to XEmacs, doing a run which generates results which can be acted upon is not necessarily a trivial task.

Also, if you're still willing to do some runs make sure you configure with the '`--quantify`' flag. That will keep Quantify from starting to record data until after the loadup is completed and will shut off recording right before it shuts down (which generates enough bogus data to throw most results off). It also enables three additional elisp commands: `quantify-start-recording-data`, `quantify-stop-recording-data` and `quantify-clear-data`.

To get started debugging XEmacs, take a look at the '`gdbinit`' and '`dbxrc`' files in the '`src`' directory. See section "Q2.1.15 - How to Debug an XEmacs problem with a debugger" in *XEmacs FAQ*.

Here are things to know when you create a new source file:

- All .c files should `#include <config.h>` first. Almost all .c files should `#include "lisp.h"` second.
- Generated header files should be included using the `<>` syntax, not the `""` syntax. The generated headers are:

  config.h puresize-adjust.h sheap-adjust.h paths.h Emacs.ad.h

  The basic rule is that you should assume builds using `--srcdir` and the `<>` syntax needs to be used when the to-be-included generated file is in a potentially different directory *at compile time*.
- Header files should not include <config.h> and "lisp.h". It is the responsibility of the .c files that use it to do so.
- If the header uses INLINE, either directly or though DECLARE_LRECORD, then it must be added to inline.c's includes.
- Try compiling at least once with

```
gcc --with-mule --with-union-type --error-checking=all
```

9 A Summary of the Various XEmacs Modules

This is accurate as of XEmacs 20.0.

9.1 Low-Level Modules

```
   size  name
-------  ---------------------
  18150  config.h
```

This is automatically generated from 'config.h.in' based on the results of configure tests and user-selected optional features and contains preprocessor definitions specifying the nature of the environment in which XEmacs is being compiled.

```
   2347  paths.h
```

This is automatically generated from 'paths.h.in' based on supplied configure values, and allows for non-standard installed configurations of the XEmacs directories. It's currently broken, though.

```
  47878  emacs.c
  20239  signal.c
```

'emacs.c' contains main() and other code that performs the most basic environment initializations and handles shutting down the XEmacs process (this includes kill-emacs, the normal way that XEmacs is exited; dump-emacs, which is used during the build process to write out the XEmacs executable; run-emacs-from-temacs, which can be used to start XEmacs directly when temacs has finished loading all the Lisp code; and emergency code to handle crashes [XEmacs tries to auto-save all files before it crashes]).

Low-level code that directly interacts with the Unix signal mechanism, however, is in 'signal.c'. Note that this code does not handle system dependencies in interfacing to signals; that is handled using the 'syssignal.h' header file, described in section J below.

```
  23458  unexaix.c
   9893  unexalpha.c
  11302  unexapollo.c
  16544  unexconvex.c
  31967  unexec.c
  30959  unexelf.c
  35791  unexelfsgi.c
   3207  unexencap.c
   7276  unexenix.c
  20539  unexfreebsd.c
   1153  unexfx2800.c
  13432  unexhp9k3.c
  11049  unexhp9k800.c
   9165  unexmips.c
   8981  unexnext.c
   1673  unexsol2.c
  19261  unexsunos4.c
```

These modules contain code dumping out the XEmacs executable on various different systems. (This process is highly machine-specific and requires intimate knowledge of the executable format and the memory map of the process.) Only one of these modules is actually used; this is chosen by '`configure`'.

```
15715  crt0.c
 1484  lastfile.c
 1115  pre-crt0.c
```

These modules are used in conjunction with the dump mechanism. On some systems, an alternative version of the C startup code (the actual code that receives control from the operating system when the process is started, and which calls `main()`) is required so that the dumping process works properly; '`crt0.c`' provides this.

'`pre-crt0.c`' and '`lastfile.c`' should be the very first and very last file linked, respectively. (Actually, this is not really true. '`lastfile.c`' should be after all Emacs modules whose initialized data should be made constant, and before all other Emacs files and all libraries. In particular, the allocation modules '`gmalloc.c`', '`alloca.c`', etc. are normally placed past '`lastfile.c`', and all of the files that implement Xt widget classes *must* be placed after '`lastfile.c`' because they contain various structures that must be statically initialized and into which Xt writes at various times.) '`pre-crt0.c`' and '`lastfile.c`' contain exported symbols that are used to determine the start and end of XEmacs' initialized data space when dumping.

```
14786  alloca.c
16678  free-hook.c
 1692  getpagesize.h
41936  gmalloc.c
25141  malloc.c
 3802  mem-limits.h
39011  ralloc.c
 3436  vm-limit.c
```

These handle basic C allocation of memory. '`alloca.c`' is an emulation of the stack allocation function `alloca()` on machines that lack this. (XEmacs makes extensive use of `alloca()` in its code.)

'`gmalloc.c`' and '`malloc.c`' are two implementations of the standard C functions `malloc()`, `realloc()` and `free()`. They are often used in place of the standard system-provided `malloc()` because they usually provide a much faster implementation, at the expense of additional memory use. '`gmalloc.c`' is a newer implementation that is much more memory-efficient for large allocations than '`malloc.c`', and should always be preferred if it works. (At one point, '`gmalloc.c`' didn't work on some systems where '`malloc.c`' worked; but this should be fixed now.)

'`ralloc.c`' is the *relocating allocator*. It provides functions similar to `malloc()`, `realloc()` and `free()` that allocate memory that can be dynamically relocated in memory. The advantage of this is that allocated memory can be shuffled around to place all the free memory at the end of the heap, and the heap can then be shrunk, releasing the memory back to the operating system. The use of this can be controlled with the configure option `--rel-alloc`; if enabled, memory allocated for buffers will be relocatable, so that if a very large file is visited and the buffer is later killed, the memory can be released to the operating system. (The disadvantage of this mechanism is that it can be very slow. On systems with the `mmap()` system call, the XEmacs version of '`ralloc.c`' uses this to move memory around without actually having to block-copy it, which can speed things up; but it can still cause noticeable performance degradation.)

'`free-hook.c`' contains some debugging functions for checking for invalid arguments to `free()`.

'`vm-limit.c`' contains some functions that warn the user when memory is getting low. These are callback functions that are called by '`gmalloc.c`' and '`malloc.c`' at appropriate times.

'`getpagesize.h`' provides a uniform interface for retrieving the size of a page in virtual memory. '`mem-limits.h`' provides a uniform interface for retrieving the total amount of available virtual memory. Both are similar in spirit to the '`sys*.h`' files described in section J, below.

```
2659  blocktype.c
1410  blocktype.h
7194  dynarr.c
2671  dynarr.h
```

These implement a couple of basic C data types to facilitate memory allocation. The `Blocktype` type efficiently manages the allocation of fixed-size blocks by minimizing the number of times that `malloc()` and `free()` are called. It allocates memory in large chunks, subdivides the chunks into blocks of the proper size, and returns the blocks as requested. When blocks are freed, they are placed onto a linked list, so they can be efficiently reused. This data type is not much used in XEmacs currently, because it's a fairly new addition.

The `Dynarr` type implements a *dynamic array*, which is similar to a standard C array but has no fixed limit on the number of elements it can contain. Dynamic arrays can hold elements of any type, and when you add a new element, the array automatically resizes itself if it isn't big enough. Dynarrs are extensively used in the redisplay mechanism.

```
2058  inline.c
```

This module is used in connection with inline functions (available in some compilers). Often, inline functions need to have a corresponding non-inline function that does the same thing. This module is where they reside. It contains no actual code, but defines some special flags that cause inline functions defined in header files to be rendered as actual functions. It then includes all header files that contain any inline function definitions, so that each one gets a real function equivalent.

```
6489  debug.c
2267  debug.h
```

These functions provide a system for doing internal consistency checks during code development. This system is not currently used; instead the simpler `assert()` macro is used along with the various checks provided by the '`--error-check-*`' configuration options.

```
1643  prefix-args.c
```

This is actually the source for a small, self-contained program used during building.

```
 904  universe.h
```

This is not currently used.

9.2 Basic Lisp Modules

```
   size  name
-------  ---------------------
  70167  emacsfns.h
   6305  lisp-disunion.h
   7086  lisp-union.h
  54929  lisp.h
  14235  lrecord.h
  10728  symsinit.h
```

These are the basic header files for all XEmacs modules. Each module includes '`lisp.h`', which brings the other header files in. '`lisp.h`' contains the definitions of the structures and extractor and constructor macros for the basic Lisp objects and various other basic definitions for the Lisp environment, as well as some general-purpose definitions (e.g. `min()` and `max()`). '`lisp.h`' includes either '`lisp-disunion.h`' or '`lisp-union.h`', depending on whether `USE_UNION_TYPE` is defined. These files define the typedef of the Lisp object itself (as described above) and the low-level macros that hide the actual implementation of the Lisp object. All extractor and constructor macros for particular types of Lisp objects are defined in terms of these low-level macros.

As a general rule, all typedefs should go into the typedefs section of '`lisp.h`' rather than into a module-specific header file even if the structure is defined elsewhere. This allows function prototypes that use the typedef to be placed into '`emacsfns.h`'. Forward structure declarations (i.e. a simple declaration like `struct foo;` where the structure itself is defined elsewhere) should be placed into the typedefs section as necessary.

'`lrecord.h`' contains the basic structures and macros that implement all record-type Lisp objects – i.e. all objects whose type is a field in their C structure, which includes all objects except the few most basic ones.

'`emacsfns.h`' contains prototypes for most of the exported functions in the various modules. (In particular, prototypes for Lisp primitives should always go into this header file. Prototypes for other functions can either go here or in a module-specific header file, depending on how general-purpose the function is and whether it has special-purpose argument types requiring definitions not in '`lisp.h`'.) All initialization functions are prototyped in '`symsinit.h`'.

```
 120478  alloc.c
   1029  pure.c
   2506  puresize.h
```

The large module '`alloc.c`' implements all of the basic allocation and garbage collection for Lisp objects. The most commonly used Lisp objects are allocated in chunks, similar to the Blocktype data type described above; others are allocated in individually `malloc()`ed blocks. This module provides the foundation on which all other aspects of the Lisp environment sit, and is the first module initialized at startup.

Note that '`alloc.c`' provides a series of generic functions that are not dependent on any particular object type, and interfaces to particular types of objects using a standardized interface of type-specific methods. This scheme is a fundamental principle of object-oriented programming and is heavily used throughout XEmacs. The great advantage of this is that it allows for a clean separation of functionality into different modules – new classes of

Lisp objects, new event interfaces, new device types, new stream interfaces, etc. can be added transparently without affecting code anywhere else in XEmacs. Because the different subsystems are divided into general and specific code, adding a new subtype within a subsystem will in general not require changes to the generic subsystem code or affect any of the other subtypes in the subsystem; this provides a great deal of robustness to the XEmacs code.

'`pure.c`' contains the declaration of the *purespace* array. Pure space is a hack used to place some constant Lisp data into the code segment of the XEmacs executable, even though the data needs to be initialized through function calls. (See above in section VIII for more info about this.) During startup, certain sorts of data is automatically copied into pure space, and other data is copied manually in some of the basic Lisp files by calling the function `purecopy`, which copies the object if possible (this only works in temacs, of course) and returns the new object. In particular, while temacs is executing, the Lisp reader automatically copies all compiled-function objects that it reads into pure space. Since compiled-function objects are large, are never modified, and typically comprise the majority of the contents of a compiled-Lisp file, this works well. While XEmacs is running, any attempt to modify an object that resides in pure space causes an error. Objects in pure space are never garbage collected – almost all of the time, they're intended to be permanent, and in any case you can't write into pure space to set the mark bits.

'`puresize.h`' contains the declaration of the size of the pure space array. This depends on the optional features that are compiled in, any extra purespace requested by the user at compile time, and certain other factors (e.g. 64-bit machines need more pure space because their Lisp objects are larger). The smallest size that suffices should be used, so that there's no wasted space. If there's not enough pure space, you will get an error during the build process, specifying how much more pure space is needed.

```
122243  eval.c
  2305  backtrace.h
```

This module contains all of the functions to handle the flow of control. This includes the mechanisms of defining functions, calling functions, traversing stack frames, and binding variables; the control primitives and other special forms such as `while`, `if`, `eval`, `let`, `and`, `or`, `progn`, etc.; handling of non-local exits, unwind-protects, and exception handlers; entering the debugger; methods for the subr Lisp object type; etc. It does *not* include the `read` function, the `print` function, or the handling of symbols and obarrays.

'`backtrace.h`' contains some structures related to stack frames and the flow of control.

```
 64949  lread.c
```

This module implements the Lisp reader and the `read` function, which converts text into Lisp objects, according to the read syntax of the objects, as described above. This is similar to the parser that is a part of all compilers.

```
 40900  print.c
```

This module implements the Lisp print mechanism and the `print` function and related functions. This is the inverse of the Lisp reader – it converts Lisp objects to a printed, textual representation. (Hopefully something that can be read back in using `read` to get an equivalent object.)

```
  4518  general.c
 60220  symbols.c
```

```
 9966  symeval.h
```

'symbols.c' implements the handling of symbols, obarrays, and retrieving the values of symbols. Much of the code is devoted to handling the special *symbol-value-magic* objects that define special types of variables – this includes buffer-local variables, variable aliases, variables that forward into C variables, etc. This module is initialized extremely early (right after 'alloc.c'), because it is here that the basic symbols `t` and `nil` are created, and those symbols are used everywhere throughout XEmacs.

'symeval.h' contains the definitions of symbol structures and the `DEFVAR_LISP()` and related macros for declaring variables.

```
48973  data.c
25694  floatfns.c
71049  fns.c
```

These modules implement the methods and standard Lisp primitives for all the basic Lisp object types other than symbols (which are described above). 'data.c' contains all the predicates (primitives that return whether an object is of a particular type); the integer arithmetic functions; and the basic accessor and mutator primitives for the various object types. 'fns.c' contains all the standard predicates for working with sequences (where, abstractly speaking, a sequence is an ordered set of objects, and can be represented by a list, string, vector, or bit-vector); it also contains `equal`, perhaps on the grounds that bulk of the operation of `equal` is comparing sequences. 'floatfns.c' contains methods and primitives for floats and floating-point arithmetic.

```
23555  bytecode.c
 3358  bytecode.h
```

'bytecode.c' implements the byte-code interpreter, and 'bytecode.h' contains associated structures. Note that the byte-code *compiler* is written in Lisp.

9.3 Modules for Standard Editing Operations

```
   size  name
-------  ---------------------
  82900  buffer.c
  60964  buffer.h
   6059  bufslots.h
```

'buffer.c' implements the *buffer* Lisp object type. This includes functions that create and destroy buffers; retrieve buffers by name or by other properties; manipulate lists of buffers (remember that buffers are permanent objects and stored in various ordered lists); retrieve or change buffer properties; etc. It also contains the definitions of all the built-in buffer-local variables (which can be viewed as buffer properties). It does *not* contain code to manipulate buffer-local variables (that's in 'symbols.c', described above); or code to manipulate the text in a buffer.

'buffer.h' defines the structures associated with a buffer and the various macros for retrieving text from a buffer and special buffer positions (e.g. `point`, the default location for text insertion). It also contains macros for working with buffer positions and converting between their representations as character offsets and as byte offsets (under MULE, they are different, because characters can be multi-byte). It is one of the largest header files.

'bufslots.h' defines the fields in the buffer structure that correspond to the built-in buffer-local variables. It is its own header file because it is included many times in 'buffer.c', as a way of iterating over all the built-in buffer-local variables.

```
   79888  insdel.c
    6103  insdel.h
```

'insdel.c' contains low-level functions for inserting and deleting text in a buffer, keeping track of changed regions for use by redisplay, and calling any before-change and after-change functions that may have been registered for the buffer. It also contains the actual functions that convert between byte offsets and character offsets.

'insdel.h' contains associated headers.

```
   10975  marker.c
```

This module implements the *marker* Lisp object type, which conceptually is a pointer to a text position in a buffer that moves around as text is inserted and deleted, so as to remain in the same relative position. This module doesn't actually move the markers around – that's handled in 'insdel.c'. This module just creates them and implements the primitives for working with them. As markers are simple objects, this does not entail much.

Note that the standard arithmetic primitives (e.g. `+`) accept markers in place of integers and automatically substitute the value of `marker-position` for the marker, i.e. an integer describing the current buffer position of the marker.

```
  193714  extents.c
   15686  extents.h
```

This module implements the *extent* Lisp object type, which is like a marker that works over a range of text rather than a single position. Extents are also much more complex and powerful than markers and have a more efficient (and more algorithmically complex) implementation. The implementation is described in detail in comments in 'extents.c'.

The code in 'extents.c' works closely with 'insdel.c' so that extents are properly moved around as text is inserted and deleted. There is also code in 'extents.c' that provides information needed by the redisplay mechanism for efficient operation. (Remember that extents can have display properties that affect [sometimes drastically, as in the `invisible` property] the display of the text they cover.)

```
   60155  editfns.c
```

'editfns.c' contains the standard Lisp primitives for working with a buffer's text, and calls the low-level functions in 'insdel.c'. It also contains primitives for working with `point` (the default buffer insertion location).

'editfns.c' also contains functions for retrieving various characteristics from the external environment: the current time, the process ID of the running XEmacs process, the name of the user who ran this XEmacs process, etc. It's not clear why this code is in 'editfns.c'.

```
   26081  callint.c
   12577  cmds.c
    2749  commands.h
```

These modules implement the basic *interactive* commands, i.e. user-callable functions. Commands, as opposed to other functions, have special ways of getting their parameters interactively (by querying the user), as opposed to having them passed in a normal function invocation. Many commands are not really meant to be called from other Lisp functions, because they modify global state in a way that's often undesired as part of other Lisp functions.

'`callint.c`' implements the mechanism for querying the user for parameters and calling interactive commands. The bulk of this module is code that parses the interactive spec that is supplied with an interactive command.

'`cmds.c`' implements the basic, most commonly used editing commands: commands to move around the current buffer and insert and delete characters. These commands are implemented using the Lisp primitives defined in '`editfns.c`'.

'`commands.h`' contains associated structure definitions and prototypes.

```
 194863  regex.c
  18968  regex.h
  79800  search.c
```

'`search.c`' implements the Lisp primitives for searching for text in a buffer, and some of the low-level algorithms for doing this. In particular, the fast fixed-string Boyer-Moore search algorithm is implemented in '`search.c`'. The low-level algorithms for doing regular-expression searching, however, are implemented in '`regex.c`' and '`regex.h`'. These two modules are largely independent of XEmacs, and are similar to (and based upon) the regular-expression routines used in '`grep`' and other GNU utilities.

```
  20476  doprnt.c
```

'`doprnt.c`' implements formatted-string processing, similar to `printf()` command in C.

```
  15372  undo.c
```

This module implements the undo mechanism for tracking buffer changes. Most of this could be implemented in Lisp.

9.4 Editor-Level Control Flow Modules

```
   size  name
-------  ---------------------
  84546  event-Xt.c
 121483  event-stream.c
   6658  event-tty.c
  49271  events.c
  14459  events.h
```

These implement the handling of events (user input and other system notifications).

'`events.c`' and '`events.h`' define the *event* Lisp object type and primitives for manipulating it.

'`event-stream.c`' implements the basic functions for working with event queues, dispatching an event by looking it up in relevant keymaps and such, and handling timeouts; this includes the primitives `next-event` and `dispatch-event`, as well as related primitives such as `sit-for`, `sleep-for`, and `accept-process-output`. ('`event-stream.c`' is one of the hairiest and trickiest modules in XEmacs. Beware! You can easily mess things up here.)

'`event-Xt.c`' and '`event-tty.c`' implement the low-level interfaces onto retrieving events from Xt (the X toolkit) and from TTY's (using `read()` and `select()`), respectively. The event interface enforces a clean separation between the specific code for interfacing with the operating system and the generic code for working with events, by defining an API of basic, low-level event methods; '`event-Xt.c`' and '`event-tty.c`' are two different implementations of this API. To add support for a new operating system (e.g. NeXTstep), one merely needs to provide another implementation of those API functions.

Note that the choice of whether to use '`event-Xt.c`' or '`event-tty.c`' is made at compile time! Or at the very latest, it is made at startup time. '`event-Xt.c`' handles events for *both* X and TTY frames; '`event-tty.c`' is only used when X support is not compiled into XEmacs. The reason for this is that there is only one event loop in XEmacs: thus, it needs to be able to receive events from all different kinds of frames.

```
 129583  keymap.c
   2621  keymap.h
```

'`keymap.c`' and '`keymap.h`' define the *keymap* Lisp object type and associated methods and primitives. (Remember that keymaps are objects that associate event descriptions with functions to be called to "execute" those events; `dispatch-event` looks up events in the relevant keymaps.)

```
  25212  keyboard.c
```

'`keyboard.c`' contains functions that implement the actual editor command loop – i.e. the event loop that cyclically retrieves and dispatches events. This code is also rather tricky, just like '`event-stream.c`'.

```
   9973  macros.c
   1397  macros.h
```

These two modules contain the basic code for defining keyboard macros. These functions don't actually do much; most of the code that handles keyboard macros is mixed in with the event-handling code in '`event-stream.c`'.

```
  23234  minibuf.c
```

This contains some miscellaneous code related to the minibuffer (most of the minibuffer code was moved into Lisp by Richard Mlynarik). This includes the primitives for completion (although filename completion is in '`dired.c`'), the lowest-level interface to the minibuffer (if the command loop were cleaned up, this too could be in Lisp), and code for dealing with the echo area (this, too, was mostly moved into Lisp, and the only code remaining is code to call out to Lisp or provide simple bootstrapping implementations early in temacs, before the echo-area Lisp code is loaded).

9.5 Modules for the Basic Displayable Lisp Objects

```
   size  name
-------  ---------------------
    985  device-ns.h
   6454  device-stream.c
   1196  device-stream.h
   9526  device-tty.c
   8660  device-tty.h
  43798  device-x.c
  11667  device-x.h
  26056  device.c
  22993  device.h
```

These modules implement the *device* Lisp object type. This abstracts a particular screen or connection on which frames are displayed. As with Lisp objects, event interfaces, and other subsystems, the device code is separated into a generic component that contains a standardized interface (in the form of a set of methods) onto particular device types.

The device subsystem defines all the methods and provides method services for not only device operations but also for the frame, window, menubar, scrollbar, toolbar, and other displayable-object subsystems. The reason for this is that all of these subsystems have the same subtypes (X, TTY, NeXTstep, Microsoft Windows, etc.) as devices do.

```
   934  frame-ns.h
  2303  frame-tty.c
 69205  frame-x.c
  5976  frame-x.h
 68175  frame.c
 15080  frame.h
```

Each device contains one or more frames in which objects (e.g. text) are displayed. A frame corresponds to a window in the window system; usually this is a top-level window but it could potentially be one of a number of overlapping child windows within a top-level window, using the MDI (Multiple Document Interface) protocol in Microsoft Windows or a similar scheme.

The '`frame-*`' files implement the *frame* Lisp object type and provide the generic and device-type-specific operations on frames (e.g. raising, lowering, resizing, moving, etc.).

```
160783  window.c
 15974  window.h
```

Each frame consists of one or more non-overlapping *windows* (better known as *panes* in standard window-system terminology) in which a buffer's text can be displayed. Windows can also have scrollbars displayed around their edges.

'`window.c`' and '`window.h`' implement the *window* Lisp object type and provide code to manage windows. Since windows have no associated resources in the window system (the window system knows only about the frame; no child windows or anything are used for XEmacs windows), there is no device-type-specific code here; all of that code is part of the redisplay mechanism or the code for particular object types such as scrollbars.

9.6 Modules for other Display-Related Lisp Objects

```
   size  name
-------  ---------------------
  54397  faces.c
  15173  faces.h

   4961  bitmaps.h
    954  glyphs-ns.h
 105345  glyphs-x.c
   4288  glyphs-x.h
  72102  glyphs.c
  16356  glyphs.h

    952  objects-ns.h
   9971  objects-tty.c
```

```
 1465  objects-tty.h
32326  objects-x.c
 2806  objects-x.h
31944  objects.c
 6809  objects.h
```

```
57511  menubar-x.c
11243  menubar.c
```

```
25012  scrollbar-x.c
 2554  scrollbar-x.h
26954  scrollbar.c
 2778  scrollbar.h
```

```
23117  toolbar-x.c
43456  toolbar.c
 4280  toolbar.h
```

```
25070  font-lock.c
```

This file provides C support for syntax highlighting – i.e. highlighting different syntactic constructs of a source file in different colors, for easy reading. The C support is provided so that this is fast.

```
32180  dgif_lib.c
 3999  gif_err.c
10697  gif_lib.h
 9371  gifalloc.c
```

These modules decode GIF-format image files, for use with glyphs.

9.7 Modules for the Redisplay Mechanism

```
   size  name
-------  ---------------------
  38692  redisplay-output.c
  40835  redisplay-tty.c
  65069  redisplay-x.c
 234142  redisplay.c
  17026  redisplay.h
```

These files provide the redisplay mechanism. As with many other subsystems in XEmacs, there is a clean separation between the general and device-specific support.

'redisplay.c' contains the bulk of the redisplay engine. These functions update the redisplay structures (which describe how the screen is to appear) to reflect any changes made to the state of any displayable objects (buffer, frame, window, etc.) since the last time that redisplay was called. These functions are highly optimized to avoid doing more work than necessary (since redisplay is called extremely often and is potentially a huge time sink), and depend heavily on notifications from the objects themselves that changes have occurred, so that redisplay doesn't explicitly have to check each possible object. The redisplay mechanism also contains a great deal of caching to further speed things up; some of this caching is contained within the various displayable objects.

'redisplay-output.c' goes through the redisplay structures and converts them into calls to device-specific methods to actually output the screen changes.

'redisplay-x.c' and 'redisplay-tty.c' are two implementations of these redisplay output methods, for X frames and TTY frames, respectively.

```
   14129  indent.c
```

This module contains various functions and Lisp primitives for converting between buffer positions and screen positions. These functions call the redisplay mechanism to do most of the work, and then examine the redisplay structures to get the necessary information. This module needs work.

```
   14754  termcap.c
    2141  terminfo.c
    7253  tparam.c
```

These files contain functions for working with the termcap (BSD-style) and terminfo (System V style) databases of terminal capabilities and escape sequences, used when XEmacs is displaying in a TTY.

```
   10869  cm.c
    5876  cm.h
```

These files provide some miscellaneous TTY-output functions and should probably be merged into 'redisplay-tty.c'.

9.8 Modules for Interfacing with the File System

```
     size  name
  -------  ---------------------
    43362  lstream.c
    14240  lstream.h
```

These modules implement the *stream* Lisp object type. This is an internal-only Lisp object that implements a generic buffering stream. The idea is to provide a uniform interface onto all sources and sinks of data, including file descriptors, stdio streams, chunks of memory, Lisp buffers, Lisp strings, etc. That way, I/O functions can be written to the stream interface and can transparently handle all possible sources and sinks. (For example, the `read` function can read data from a file, a string, a buffer, or even a function that is called repeatedly to return data, without worrying about where the data is coming from or what-size chunks it is returned in.)

Note that in the C code, streams are called *lstreams* (for "Lisp streams") to distinguish them from other kinds of streams, e.g. stdio streams and C++ I/O streams.

Similar to other subsystems in XEmacs, lstreams are separated into generic functions and a set of methods for the different types of lstreams. 'lstream.c' provides implementations of many different types of streams; others are provided, e.g., in 'mule-coding.c'.

```
   126926  fileio.c
```

This implements the basic primitives for interfacing with the file system. This includes primitives for reading files into buffers, writing buffers into files, checking for the presence or accessibility of files, canonicalizing file names, etc. Note that these primitives are usually not invoked directly by the user: There is a great deal of higher-level Lisp code that implements the user commands such as `find-file` and `save-buffer`. This is similar to the distinction between the lower-level primitives in 'editfns.c' and the higher-level user commands in 'commands.c' and 'simple.el'.

```
   10960  filelock.c
```

This file provides functions for detecting clashes between different processes (e.g. XEmacs and some external process, or two different XEmacs processes) modifying the same file. (XEmacs can optionally use the 'lock/' subdirectory to provide a form of "locking" between different XEmacs processes.) This module is also used by the low-level functions in 'insdel.c' to ensure that, if the first modification is being made to a buffer whose corresponding file has been externally modified, the user is made aware of this so that the buffer can be synched up with the external changes if necessary.

```
    4527  filemode.c
```

This file provides some miscellaneous functions that construct a 'rwxr-xr-x'-type permissions string (as might appear in an 'ls'-style directory listing) given the information returned by the `stat()` system call.

```
   22855  dired.c
    2094  ndir.h
```

These files implement the XEmacs interface to directory searching. This includes a number of primitives for determining the files in a directory and for doing filename completion. (Remember that generic completion is handled by a different mechanism, in 'minibuf.c'.)

'ndir.h' is a header file used for the directory-searching emulation functions provided in 'sysdep.c' (see section J below), for systems that don't provide any directory-searching functions. (On those systems, directories can be read directly as files, and parsed.)

```
    4311  realpath.c
```

This file provides an implementation of the `realpath()` function for expanding symbolic links, on systems that don't implement it or have a broken implementation.

9.9 Modules for Other Aspects of the Lisp Interpreter and Object System

```
     size  name
  -------  ---------------------
    22290  elhash.c
     2454  elhash.h
    12169  hash.c
     3369  hash.h
```

These files implement the *hashtable* Lisp object type. 'hash.c' and 'hash.h' provide a generic C implementation of hash tables (which can stand independently of XEmacs), and 'elhash.c' and 'elhash.h' provide a Lisp interface onto the C hash tables using the hashtable Lisp object type.

```
    95691  specifier.c
    11167  specifier.h
```

This module implements the *specifier* Lisp object type. This is primarily used for displayable properties, and allows for values that are specific to a particular buffer, window, frame, device, or device class, as well as a default value existing. This is used, for example, to control the height of the horizontal scrollbar or the appearance of the `default`, `bold`, or other faces. The specifier object consists of a number of specifications, each of which maps from a buffer, window, etc. to a value. The function `specifier-instance` looks up a value given a window (from which a buffer, frame, and device can be derived).

```
43058  chartab.c
 6503  chartab.h
 9918  casetab.c
```

'`chartab.c`' and '`chartab.h`' implement the *char table* Lisp object type, which maps from characters or certain sorts of character ranges to Lisp objects. The implementation of this object type is optimized for the internal representation of characters. Char tables come in different types, which affect the allowed object types to which a character can be mapped and also dictate certain other properties of the char table.

'`casetab.c`' implements one sort of char table, the *case table*, which maps characters to other characters of possibly different case. These are used by XEmacs to implement case-changing primitives and to do case-insensitive searching.

```
49593  syntax.c
10200  syntax.h
```

This module implements *syntax tables*, another sort of char table that maps characters into syntax classes that define the syntax of these characters (e.g. a parenthesis belongs to a class of '`open`' characters that have corresponding '`close`' characters and can be nested). This module also implements the Lisp *scanner*, a set of primitives for scanning over text based on syntax tables. This is used, for example, to find the matching parenthesis in a command such as `forward-sexp`, and by '`font-lock.c`' to locate quoted strings, comments, etc.

```
10438  casefiddle.c
```

This module implements various Lisp primitives for upcasing, downcasing and capitalizing strings or regions of buffers.

```
20234  rangetab.c
```

This module implements the *range table* Lisp object type, which provides for a mapping from ranges of integers to arbitrary Lisp objects.

```
 3201  opaque.c
 2206  opaque.h
```

This module implements the *opaque* Lisp object type, an internal-only Lisp object that encapsulates an arbitrary block of memory so that it can be managed by the Lisp allocation system. To create an opaque object, you call `make_opaque()`, passing a pointer to a block of memory. An object is created that is big enough to hold the memory, which is copied into the object's storage. The object will then stick around as long as you keep pointers to it, after which it will be automatically reclaimed.

Opaque objects can also have an arbitrary *mark method* associated with them, in case the block of memory contains other Lisp objects that need to be marked for garbage-collection purposes. (If you need other object methods, such as a finalize method, you should just go ahead and create a new Lisp object type – it's not hard.)

```
 8783  abbrev.c
```

This function provides a few primitives for doing dynamic abbreviation expansion. In XEmacs, most of the code for this has been moved into Lisp. Some C code remains for speed and because the primitive `self-insert-command` (which is executed for all self-inserting characters) hooks into the abbrev mechanism. (`self-insert-command` is itself in C only for speed.)

```
   21934  doc.c
```

This function provides primitives for retrieving the documentation strings of functions and variables. These documentation strings contain certain special markers that get dynamically expanded (e.g. a reverse-lookup is performed on some named functions to retrieve their current key bindings). Some documentation strings (in particular, for the built-in primitives and pre-loaded Lisp functions) are stored externally in a file '`DOC`' in the '`lib-src/`' directory and need to be fetched from that file. (Part of the build stage involves building this file, and another part involves constructing an index for this file and embedding it into the executable, so that the functions in '`doc.c`' do not have to search the entire '`DOC`' file to find the appropriate documentation string.)

```
   13197  md5.c
```

This function provides a Lisp primitive that implements the MD5 secure hashing scheme, used to create a large hash value of a string of data such that the data cannot be derived from the hash value. This is used for various security applications on the Internet.

9.10 Modules for Interfacing with the Operating System

```
     size  name
  -------  ---------------------
    33533  callproc.c
    89697  process.c
     4663  process.h
```

These modules allow XEmacs to spawn and communicate with subprocesses and network connections.

'`callproc.c`' implements (through the `call-process` primitive) what are called *synchronous subprocesses*. This means that XEmacs runs a program, waits till it's done, and retrieves its output. A typical example might be calling the '`ls`' program to get a directory listing.

'`process.c`' and '`process.h`' implement *asynchronous subprocesses*. This means that XEmacs starts a program and then continues normally, not waiting for the process to finish. Data can be sent to the process or retrieved from it as it's running. This is used for the `shell` command (which provides a front end onto a shell program such as '`csh`'), the mail and news readers implemented in XEmacs, etc. The result of calling `start-process` to start a subprocess is a process object, a particular kind of object used to communicate with the subprocess. You can send data to the process by passing the process object and the data to `send-process`, and you can specify what happens to data retrieved from the process by setting properties of the process object. (When the process sends data, XEmacs receives a process event, which says that there is data ready. When `dispatch-event` is called on this event, it reads the data from the process and does something with it, as specified by the process object's properties. Typically, this means inserting the data into a buffer or calling a function.) Another property of the process object is called the *sentinel*, which is a function that is called when the process terminates.

Process objects are also used for network connections (connections to a process running on another machine). Network connections are started with `open-network-stream` but otherwise work just like subprocesses.

```
136029  sysdep.c
  5986  sysdep.h
```

These modules implement most of the low-level, messy operating-system interface code. This includes various device control (ioctl) operations for file descriptors, TTY's, pseudo-terminals, etc. (usually this stuff is fairly system-dependent; thus the name of this module), and emulation of standard library functions and system calls on systems that don't provide them or have broken versions.

```
 3605  sysdir.h
 6708  sysfile.h
 2027  sysfloat.h
 2918  sysproc.h
  745  syspwd.h
 7643  syssignal.h
 6892  systime.h
12477  systty.h
 3487  syswait.h
```

These header files provide consistent interfaces onto system-dependent header files and system calls. The idea is that, instead of including a standard header file like '`<sys/param.h>`' (which may or may not exist on various systems) or having to worry about whether all system provide a particular preprocessor constant, or having to deal with the four different paradigms for manipulating signals, you just include the appropriate '`sys*.h`' header file, which includes all the right system header files, defines and missing preprocessor constants, provides a uniform interface onto system calls, etc.

'`sysdir.h`' provides a uniform interface onto directory-querying functions. (In some cases, this is in conjunction with emulation functions in '`sysdep.c`'.)

'`sysfile.h`' includes all the necessary header files for standard system calls (e.g. `read()`), ensures that all necessary `open()` and `stat()` preprocessor constants are defined, and possibly (usually) substitutes sugared versions of `read()`, `write()`, etc. that automatically restart interrupted I/O operations.

'`sysfloat.h`' includes the necessary header files for floating-point operations.

'`sysproc.h`' includes the necessary header files for calling `select()`, `fork()`, `execve()`, socket operations, and the like, and ensures that the `FD_*()` macros for descriptor-set manipulations are available.

'`syspwd.h`' includes the necessary header files for obtaining information from '`/etc/passwd`' (the functions are emulated under VMS).

'`syssignal.h`' includes the necessary header files for signal-handling and provides a uniform interface onto the different signal-handling and signal-blocking paradigms.

'`systime.h`' includes the necessary header files and provides uniform interfaces for retrieving the time of day, setting file access/modification times, getting the amount of time used by the XEmacs process, etc.

'`systty.h`' buffers against the infinitude of different ways of controlling TTY's.

'syswait.h' provides a uniform way of retrieving the exit status from a wait()ed-on process (some systems use a union, others use an int).

```
 7940  hpplay.c
10920  libsst.c
 1480  libsst.h
 3260  libst.h
15355  linuxplay.c
15849  nas.c
19133  sgiplay.c
15411  sound.c
 7358  sunplay.c
```

These files implement the ability to play various sounds on some types of computers. You have to configure your XEmacs with sound support in order to get this capability.

'sound.c' provides the generic interface. It implements various Lisp primitives and variables that let you specify which sounds should be played in certain conditions. (The conditions are identified by symbols, which are passed to ding to make a sound. Various standard functions call this function at certain times; if sound support does not exist, a simple beep results.

'sgiplay.c', 'sunplay.c', 'hpplay.c', and 'linuxplay.c' interface to the machine's speaker for various different kind of machines. This is called *native* sound.

'nas.c' interfaces to a computer somewhere else on the network using the NAS (Network Audio Server) protocol, playing sounds on that machine. This allows you to run XEmacs on a remote machine, with its display set to your local machine, and have the sounds be made on your local machine, provided that you have a NAS server running on your local machine.

'libsst.c', 'libsst.h', and 'libst.h' provide some additional functions for playing sound on a Sun SPARC but are not currently in use.

```
44368  tooltalk.c
 2137  tooltalk.h
```

These two modules implement an interface to the ToolTalk protocol, which is an interprocess communication protocol implemented on some versions of Unix. ToolTalk is a high-level protocol that allows processes to register themselves as providers of particular services; other processes can then request a service without knowing or caring exactly who is providing the service. It is similar in spirit to the DDE protocol provided under Microsoft Windows. ToolTalk is a part of the new CDE (Common Desktop Environment) specification and is used to connect the parts of the SPARCWorks development environment.

```
22695  getloadavg.c
```

This module provides the ability to retrieve the system's current load average. (The way to do this is highly system-specific, unfortunately, and requires a lot of special-case code.)

```
148520  energize.c
  6896  energize.h
```

This module provides code to interface to an Energize server (when XEmacs is used as part of Lucid's Energize development environment) and provides some other Energize-specific functions. Much of the code in this module should be made more general-purpose and moved elsewhere, but is no longer very relevant now that Lucid is defunct. It also hasn't worked since version 19.12, since nobody has been maintaining it.

```
   2861  sunpro.c
```

This module provides a small amount of code used internally at Sun to keep statistics on the usage of XEmacs.

```
   5548  broken-sun.h
   3468  strcmp.c
   2179  strcpy.c
   1650  sunOS-fix.c
```

These files provide replacement functions and prototypes to fix numerous bugs in early releases of SunOS 4.1.

```
  11669  hftctl.c
```

This module provides some terminal-control code necessary on versions of AIX prior to 4.1.

```
   1776  acldef.h
   1602  chpdef.h
   9032  uaf.h
    105  vlimit.h
   7145  vms-pp.c
   1158  vms-pwd.h
  26532  vmsfns.c
   6038  vmsmap.c
    695  vmspaths.h
  17482  vmsproc.c
    469  vmsproc.h
```

All of these files are used for VMS support, which has never worked in XEmacs.

```
  28316  msdos.c
   1472  msdos.h
```

These modules are used for MS-DOS support, which does not work in XEmacs.

9.11 Modules for Interfacing with X Windows

```
   size  name
-------  ---------------------
   3196  Emacs.ad.h
```

A file generated from 'Emacs.ad', which contains XEmacs-supplied fallback resources (so that XEmacs has pretty defaults).

```
  24242  EmacsFrame.c
   6979  EmacsFrame.h
   3351  EmacsFrameP.h
```

These modules implement an Xt widget class that encapsulates a frame. This is for ease in integrating with Xt. The EmacsFrame widget covers the entire X window except for the menubar; the scrollbars are positioned on top of the EmacsFrame widget.

Warning: Abandon hope, all ye who enter here. This code took an ungodly amount of time to get right, and is likely to fall apart mercilessly at the slightest change. Such is life under Xt.

```
 8178  EmacsManager.c
 1967  EmacsManager.h
 1895  EmacsManagerP.h
```

These modules implement a simple Xt manager (i.e. composite) widget class that simply lets its children set whatever geometry they want. It's amazing that Xt doesn't provide this standardly, but on second thought, it makes sense, considering how amazingly broken Xt is.

```
13188  EmacsShell-sub.c
 4588  EmacsShell.c
 2180  EmacsShell.h
 3133  EmacsShellP.h
```

These modules implement two Xt widget classes that are subclasses of the TopLevelShell and TransientShell classes. This is necessary to deal with more brokenness that Xt has sadistically thrust onto the backs of developers.

```
 9673  xgccache.c
 1111  xgccache.h
```

These modules provide functions for maintenance and caching of GC's (graphics contexts) under the X Window System. This code is junky and needs to be rewritten.

```
69181  xselect.c
```

This module provides an interface to the X Window System's concept of *selections*, the standard way for X applications to communicate with each other.

```
  929  xintrinsic.h
 1038  xintrinsicp.h
 1579  xmmanagerp.h
 1585  xmprimitivep.h
```

These header files are similar in spirit to the 'sys*.h' files and buffer against different implementations of Xt and Motif.

- 'xintrinsic.h' should be included in place of '<Intrinsic.h>'.
- 'xintrinsicp.h' should be included in place of '<IntrinsicP.h>'.
- 'xmmanagerp.h' should be included in place of '<XmManagerP.h>'.
- 'xmprimitivep.h' should be included in place of '<XmPrimitiveP.h>'.

```
16930  xmu.c
  936  xmu.h
```

These files provide an emulation of the Xmu library for those systems (i.e. HPUX) that don't provide it as a standard part of X.

```
 4201  ExternalClient-Xlib.c
18083  ExternalClient.c
 2035  ExternalClient.h
 2104  ExternalClientP.h
22684  ExternalShell.c
 1709  ExternalShell.h
 1971  ExternalShellP.h
 2478  extw-Xlib.c
 1481  extw-Xlib.h
 6565  extw-Xt.c
```

```
    1430  extw-Xt.h
```

These files provide the *external widget* interface, which allows an XEmacs frame to appear as a widget in another application. To do this, you have to configure with '--external-widget'.

'ExternalShell*' provides the server (XEmacs) side of the connection.

'ExternalClient*' provides the client (other application) side of the connection. These files are not compiled into XEmacs but are compiled into libraries that are then linked into your application.

'extw-*' is common code that is used for both the client and server.

Don't touch this code; something is liable to break if you do.

```
   31014  epoch.c
```

This file provides some additional, Epoch-compatible, functionality for interfacing to the X Window System.

9.12 Modules for Internationalization

```
    size  name
 -------  ---------------------
   42836  mule-canna.c
   16737  mule-ccl.c
   41080  mule-charset.c
   30176  mule-charset.h
  146844  mule-coding.c
   16588  mule-coding.h
    6996  mule-mcpath.c
    2899  mule-mcpath.h
   57158  mule-wnnfns.c
    3351  mule.c
```

These files implement the MULE (Asian-language) support. Note that MULE actually provides a general interface for all sorts of languages, not just Asian languages (although they are generally the most complicated to support). This code is still in beta.

'mule-charset.*' and 'mule-coding.*' provide the heart of the XEmacs MULE support. 'mule-charset.*' implements the *charset* Lisp object type, which encapsulates a character set (an ordered one- or two-dimensional set of characters, such as US ASCII or JISX0208 Japanese Kanji).

'mule-coding.*' implements the *coding-system* Lisp object type, which encapsulates a method of converting between different encodings. An encoding is a representation of a stream of characters, possibly from multiple character sets, using a stream of bytes or words, and defines (e.g.) which escape sequences are used to specify particular character sets, how the indices for a character are converted into bytes (sometimes this involves setting the high bit; sometimes complicated rearranging of the values takes place, as in the Shift-JIS encoding), etc.

'mule-ccl.c' provides the CCL (Code Conversion Language) interpreter. CCL is similar in spirit to Lisp byte code and is used to implement converters for custom encodings.

'`mule-canna.c`' and '`mule-wnnfns.c`' implement interfaces to external programs used to implement the Canna and WNN input methods, respectively. This is currently in beta.

'`mule-mcpath.c`' provides some functions to allow for pathnames containing extended characters. This code is fragmentary, obsolete, and completely non-working. Instead, *pathname-coding-system* is used to specify conversions of names of files and directories. The standard C I/O functions like '`open()`' are wrapped so that conversion occurs automatically.

'`mule.c`' provides a few miscellaneous things that should probably be elsewhere.

```
9400  intl.c
```

This provides some miscellaneous internationalization code for implementing message translation and interfacing to the Ximp input method. None of this code is currently working.

```
1764  iso-wide.h
```

This contains leftover code from an earlier implementation of Asian-language support, and is not currently used.

10 Allocation of Objects in XEmacs Lisp

10.1 Introduction to Allocation

Emacs Lisp, like all Lisps, has garbage collection. This means that the programmer never has to explicitly free (destroy) an object; it happens automatically when the object becomes inaccessible. Most experts agree that garbage collection is a necessity in a modern, high-level language. Its omission from C stems from the fact that C was originally designed to be a nice abstract layer on top of assembly language, for writing kernels and basic system utilities rather than large applications.

Lisp objects can be created by any of a number of Lisp primitives. Most object types have one or a small number of basic primitives for creating objects. For conses, the basic primitive is `cons`; for vectors, the primitives are `make-vector` and `vector`; for symbols, the primitives are `make-symbol` and `intern`; etc. Some Lisp objects, especially those that are primarily used internally, have no corresponding Lisp primitives. Every Lisp object, though, has at least one C primitive for creating it.

Recall from section (VII) that a Lisp object, as stored in a 32-bit or 64-bit word, has a mark bit, a few tag bits, and a "value" that occupies the remainder of the bits. We can separate the different Lisp object types into four broad categories:

- (a) Those for whom the value directly represents the contents of the Lisp object. Only two types are in this category: integers and characters. No special allocation or garbage collection is necessary for such objects. Lisp objects of these types do not need to be `GCPRO`ed.

In the remaining three categories, the value is a pointer to a structure.

- (b) Those for whom the tag directly specifies the type. Recall that there are only three tag bits; this means that at most five types can be specified this way. The most commonly-used types are stored in this format; this includes conses, strings, vectors, and sometimes symbols. With the exception of vectors, objects in this category are allocated in *frob blocks*, i.e. large blocks of memory that are subdivided into individual objects. This saves a lot on malloc overhead, since there are typically quite a lot of these objects around, and the objects are small. (A cons, for example, occupies 8 bytes on 32-bit machines – 4 bytes for each of the two objects it contains.) Vectors are individually `malloc()`ed since they are of variable size. (It would be possible, and desirable, to allocate vectors of certain small sizes out of frob blocks, but it isn't currently done.) Strings are handled specially: Each string is allocated in two parts, a fixed size structure containing a length and a data pointer, and the actual data of the string. The former structure is allocated in frob blocks as usual, and the latter data is stored in *string chars blocks* and is relocated during garbage collection to eliminate holes.

In the remaining two categories, the type is stored in the object itself. The tag for all such objects is the generic *lrecord* (Lisp_Record) tag. The first four bytes (or eight, for 64-bit machines) of the object's structure are a pointer to a structure that describes the object's type, which includes method pointers and a pointer to a string naming the type. Note that it's possible to save some space by using a one- or two-byte tag, rather than a four- or eight-byte pointer to store the type, but it's not clear it's worth making the change.

- (c) Those lrecords that are allocated in frob blocks (see above). This includes the objects that are most common and relatively small, and includes floats, bytecodes, symbols (when not in category (b)), extents, events, and markers. With the cleanup of frob blocks done in 19.12, it's not terribly hard to add more objects to this category, but it's a bit trickier than adding an object type to type (d) (esp. if the object needs a finalization method), and is not likely to save much space unless the object is small and there are many of them. (In fact, if there are very few of them, it might actually waste space.)
- (d) Those lrecords that are individually `malloc()`ed. These are called *lcrecords*. All other types are in this category. Adding a new type to this category is comparatively easy, and all types added since 19.8 (when the current allocation scheme was devised, by Richard Mlynarik), with the exception of the character type, have been in this category.

Note that bit vectors are a bit of a special case. They are simple lrecords as in category (c), but are individually `malloc()`ed like vectors. You can basically view them as exactly like vectors except that their type is stored in lrecord fashion rather than in directly-tagged fashion.

Note that FSF Emacs redesigned their object system in 19.29 to follow a similar scheme. However, given RMS's expressed dislike for data abstraction, the FSF scheme is not nearly as clean or as easy to extend. (FSF calls items of type (c) `Lisp_Misc` and items of type (d) `Lisp_Vectorlike`, with separate tags for each, although `Lisp_Vectorlike` is also used for vectors.)

10.2 Garbage Collection

Garbage collection is simple in theory but tricky to implement. Emacs Lisp uses the oldest garbage collection method, called *mark and sweep*. Garbage collection begins by starting with all accessible locations (i.e. all variables and other slots where Lisp objects might occur) and recursively traversing all objects accessible from those slots, marking each one that is found. We then go through all of memory and free each object that is not marked, and unmarking each object that is marked. Note that "all of memory" means all currently allocated objects. Traversing all these objects means traversing all frob blocks, all vectors (which are chained in one big list), and all lcrecords (which are likewise chained).

Note that, when an object is marked, the mark has to occur inside of the object's structure, rather than in the 32-bit `Lisp_Object` holding the object's pointer; i.e. you can't just set the pointer's mark bit. This is because there may be many pointers to the same object. This means that the method of marking an object can differ depending on the type. The different marking methods are approximately as follows:

1. For conses, the mark bit of the car is set.
2. For strings, the mark bit of the string's plist is set.
3. For symbols when not lrecords, the mark bit of the symbol's plist is set.
4. For vectors, the length is negated after adding 1.
5. For lrecords, the pointer to the structure describing the type is changed (see below).
6. Integers and characters do not need to be marked, since no allocation occurs for them.

The details of this are in the `mark_object()` function.

Note that any code that operates during garbage collection has to be especially careful because of the fact that some objects may be marked and as such may not look like they normally do. In particular:

Some object pointers may have their mark bit set. This will make `FOOBARP()` predicates fail. Use `GC_FOOBARP()` to deal with this.

- Even if you clear the mark bit, `FOOBARP()` will still fail for lrecords because the implementation pointer has been changed (see below). `GC_FOOBARP()` will correctly deal with this.
- Vectors have their size field munged, so anything that looks at this field will fail.
- Note that `XFOOBAR()` macros *will* work correctly on object pointers with their mark bit set, because the logical shift operations that remove the tag also remove the mark bit.

Finally, note that garbage collection can be invoked explicitly by calling `garbage-collect` but is also called automatically by `eval`, once a certain amount of memory has been allocated since the last garbage collection (according to `gc-cons-threshold`).

10.3 GCPROing

`GCPRO`ing is one of the ugliest and trickiest parts of Emacs internals. The basic idea is that whenever garbage collection occurs, all in-use objects must be reachable somehow or other from one of the roots of accessibility. The roots of accessibility are:

1. All objects that have been `staticpro()`d. This is used for any global C variables that hold Lisp objects. A call to `staticpro()` happens implicitly as a result of any symbols declared with `defsymbol()` and any variables declared with `DEFVAR_FOO()`. You need to explicitly call `staticpro()` (in the `vars_of_foo()` method of a module) for other global C variables holding Lisp objects. (This typically includes internal lists and such things.)

   Note that `obarray` is one of the `staticpro()`d things. Therefore, all functions and variables get marked through this.
2. Any shadowed bindings that are sitting on the `specpdl` stack.
3. Any objects sitting in currently active (Lisp) stack frames, catches, and condition cases.
4. A couple of special-case places where active objects are located.
5. Anything currently marked with `GCPRO`.

Marking with `GCPRO` is necessary because some C functions (quite a lot, in fact), allocate objects during their operation. Quite frequently, there will be no other pointer to the object while the function is running, and if a garbage collection occurs and the object needs to be referenced again, bad things will happen. The solution is to mark those objects with `GCPRO`. Unfortunately this is easy to forget, and there is basically no way around this problem. Here are some rules, though:

1. For every `GCPRO`*n*, there have to be declarations of `struct gcpro gcpro1, gcpro2`, etc.
2. You *must* `UNGCPRO` anything that's `GCPRO`ed, and you *must not* `UNGCPRO` if you haven't `GCPRO`ed. Getting either of these wrong will lead to crashes, often in completely random places unrelated to where the problem lies.

3. The way this actually works is that all currently active `GCPRO`s are chained through the `struct gcpro` local variables, with the variable '`gcprolist`' pointing to the head of the list and the nth local `gcpro` variable pointing to the first `gcpro` variable in the next enclosing stack frame. Each `GCPRO`ed thing is an lvalue, and the `struct gcpro` local variable contains a pointer to this lvalue. This is why things will mess up badly if you don't pair up the `GCPRO`s and `UNGCPRO`s – you will end up with `gcprolist`s containing pointers to `struct gcpro`s or local `Lisp_Object` variables in no-longer-active stack frames.
4. It is actually possible for a single `struct gcpro` to protect a contiguous array of any number of values, rather than just a single lvalue. To effect this, call `GCPRO`*n* as usual on the first object in the array and then set `gcpro`*n*`.nvars`.
5. **Strings are relocated.** What this means in practice is that the pointer obtained using `XSTRING_DATA()` is liable to change at any time, and you should never keep it around past any function call, or pass it as an argument to any function that might cause a garbage collection. This is why a number of functions accept either a "non-relocatable" `char *` pointer or a relocatable Lisp string, and only access the Lisp string's data at the very last minute. In some cases, you may end up having to `alloca()` some space and copy the string's data into it.
6. By convention, if you have to nest `GCPRO`'s, use `NGCPRO`*n* (along with `struct gcpro ngcpro1, ngcpro2`, etc.), `NNGCPRO`*n*, etc. This avoids compiler warnings about shadowed locals.
7. It is *always* better to err on the side of extra `GCPRO`s rather than too few. The extra cycles spent on this are almost never going to make a whit of difference in the speed of anything.
8. The general rule to follow is that caller, not callee, `GCPRO`s. That is, you should not have to explicitly `GCPRO` any Lisp objects that are passed in as parameters.

   One exception from this rule is if you ever plan to change the parameter value, and store a new object in it. In that case, you *must* `GCPRO` the parameter, because otherwise the new object will not be protected.

   So, if you create any Lisp objects (remember, this happens in all sorts of circumstances, e.g. with `Fcons()`, etc.), you are responsible for `GCPRO`ing them, unless you are *absolutely sure* that there's no possibility that a garbage-collection can occur while you need to use the object. Even then, consider `GCPRO`ing.
9. A garbage collection can occur whenever anything calls `Feval`, or whenever a QUIT can occur where execution can continue past this. (Remember, this is almost anywhere.)
10. If you have the *least smidgeon of doubt* about whether you need to `GCPRO`, you should `GCPRO`.
11. Beware of `GCPRO`ing something that is uninitialized. If you have any shade of doubt about this, initialize all your variables to `Qnil`.
12. Be careful of traps, like calling `Fcons()` in the argument to another function. By the "caller protects" law, you should be `GCPRO`ing the newly-created cons, but you aren't. A certain number of functions that are commonly called on freshly created stuff (e.g. `nconc2()`, `Fsignal()`), break the "caller protects" law and go ahead and `GCPRO` their arguments so as to simplify things, but make sure and check if it's OK whenever doing something like this.

13. Once again, remember to `GCPRO`! Bugs resulting from insufficient `GCPRO`ing are intermittent and extremely difficult to track down, often showing up in crashes inside of `garbage-collect` or in weirdly corrupted objects or even in incorrect values in a totally different section of code.

Given the extremely error-prone nature of the `GCPRO` scheme, and the difficulties in tracking down, it should be considered a deficiency in the XEmacs code. A solution to this problem would involve implementing so-called *conservative* garbage collection for the C stack. That involves looking through all of stack memory and treating anything that looks like a reference to an object as a reference. This will result in a few objects not getting collected when they should, but it obviates the need for `GCPRO`ing, and allows garbage collection to happen at any point at all, such as during object allocation.

10.4 Integers and Characters

Integer and character Lisp objects are created from integers using the macros `XSETINT()` and `XSETCHAR()` or the equivalent functions `make_int()` and `make_char()`. (These are actually macros on most systems.) These functions basically just do some moving of bits around, since the integral value of the object is stored directly in the `Lisp_Object`.

`XSETINT()` and the like will truncate values given to them that are too big; i.e. you won't get the value you expected but the tag bits will at least be correct.

10.5 Allocation from Frob Blocks

The uninitialized memory required by a `Lisp_Object` of a particular type is allocated using `ALLOCATE_FIXED_TYPE()`. This only occurs inside of the lowest-level object-creating functions in '`alloc.c`': `Fcons()`, `make_float()`, `Fmake_byte_code()`, `Fmake_symbol()`, `allocate_extent()`, `allocate_event()`, `Fmake_marker()`, and `make_uninit_string()`. The idea is that, for each type, there are a number of frob blocks (each 2K in size); each frob block is divided up into object-sized chunks. Each frob block will have some of these chunks that are currently assigned to objects, and perhaps some that are free. (If a frob block has nothing but free chunks, it is freed at the end of the garbage collection cycle.) The free chunks are stored in a free list, which is chained by storing a pointer in the first four bytes of the chunk. (Except for the free chunks at the end of the last frob block, which are handled using an index which points past the end of the last-allocated chunk in the last frob block.) `ALLOCATE_FIXED_TYPE()` first tries to retrieve a chunk from the free list; if that fails, it calls `ALLOCATE_FIXED_TYPE_FROM_BLOCK()`, which looks at the end of the last frob block for space, and creates a new frob block if there is none. (There are actually two versions of these macros, one of which is more defensive but less efficient and is used for error-checking.)

10.6 lrecords

[see '`lrecord.h`']

All lrecords have at the beginning of their structure a `struct lrecord_header`. This just contains a pointer to a `struct lrecord_implementation`, which is a structure containing method pointers and such. There is one of these for each type, and it is a global, constant, statically-declared structure that is declared in the `DEFINE_LRECORD_IMPLEMENTATION()` macro. (This macro actually declares an array of two `struct lrecord_implementation` structures. The first one contains all the standard method pointers, and is used in all normal circumstances. During garbage collection, however, the lrecord is *marked* by bumping its implementation pointer by one, so that it points to the second structure in the array. This structure contains a special indication in it that it's a *marked-object* structure: the finalize method is the special function `this_marks_a_marked_record()`, and all other methods are null pointers. At the end of garbage collection, all lrecords will either be reclaimed or unmarked by decrementing their implementation pointers, so this second structure pointer will never remain past garbage collection.

Simple lrecords (of type (c) above) just have a `struct lrecord_header` at their beginning. lcrecords, however, actually have a `struct lcrecord_header`. This, in turn, has a `struct lrecord_header` at its beginning, so sanity is preserved; but it also has a pointer used to chain all lcrecords together, and a special ID field used to distinguish one lcrecord from another. (This field is used only for debugging and could be removed, but the space gain is not significant.)

Simple lrecords are created using `ALLOCATE_FIXED_TYPE()`, just like for other frob blocks. The only change is that the implementation pointer must be initialized correctly. (The implementation structure for an lrecord, or rather the pointer to it, is named `lrecord_float`, `lrecord_extent`, `lrecord_buffer`, etc.)

lcrecords are created using `alloc_lcrecord()`. This takes a size to allocate and an implementation pointer. (The size needs to be passed because some lcrecords, such as window configurations, are of variable size.) This basically just `malloc()`s the storage, initializes the `struct lcrecord_header`, and chains the lcrecord onto the head of the list of all lcrecords, which is stored in the variable `all_lcrecords`. The calls to `alloc_lcrecord()` generally occur in the lowest-level allocation function for each lrecord type.

Whenever you create an lrecord, you need to call either `DEFINE_LRECORD_IMPLEMENTATION()` or `DEFINE_LRECORD_SEQUENCE_IMPLEMENTATION()`. This needs to be specified in a C file, at the top level. What this actually does is define and initialize the implementation structure for the lrecord. (And possibly declares a function `error_check_foo()` that implements the `XFOO()` macro when error-checking is enabled.) The arguments to the macros are the actual type name (this is used to construct the C variable name of the lrecord implementation structure and related structures using the '`##`' macro concatenation operator), a string that names the type on the Lisp level (this may not be the same as the C type name; typically, the C type name has underscores, while the Lisp string has dashes), various method pointers, and the name of the C structure that contains the object. The methods are used to encapsulate type-specific information about the object, such as how to print it or mark it for garbage collection, so that it's easy to add new object types without having to add a specific case for each new type in a bunch of different places.

The difference between `DEFINE_LRECORD_IMPLEMENTATION()` and `DEFINE_LRECORD_SEQUENCE_IMPLEMENTATION()` is that the former is used for fixed-size object types and the latter is for variable-size object types. Most object types are fixed-size; some complex types, however (e.g. window configurations), are variable-size. Variable-size object types have an extra method, which is called to determine the actual size of a particular object of that type. (Currently this is only used for keeping allocation statistics.)

For the purpose of keeping allocation statistics, the allocation engine keeps a list of all the different types that exist. Note that, since `DEFINE_LRECORD_IMPLEMENTATION()` is a macro that is specified at top-level, there is no way for it to add to the list of all existing types. What happens instead is that each implementation structure contains in it a dynamically assigned number that is particular to that type. (Or rather, it contains a pointer to another structure that contains this number. This evasiveness is done so that the implementation structure can be declared const.) In the sweep stage of garbage collection, each lrecord is examined to see if its implementation structure has its dynamically-assigned number set. If not, it must be a new type, and it is added to the list of known types and a new number assigned. The number is used to index into an array holding the number of objects of each type and the total memory allocated for objects of that type. The statistics in this array are also computed during the sweep stage. These statistics are returned by the call to `garbage-collect` and are printed out at the end of the loadup phase.

Note that for every type defined with a `DEFINE_LRECORD_*()` macro, there needs to be a `DECLARE_LRECORD_IMPLEMENTATION()` somewhere in a '`.h`' file, and this '`.h`' file needs to be included by '`inline.c`'.

Furthermore, there should generally be a set of `XFOOBAR()`, `FOOBARP()`, etc. macros in a '`.h`' (or occasionally '`.c`') file. To create one of these, copy an existing model and modify as necessary.

The various methods in the lrecord implementation structure are:

1. A *mark* method. This is called during the marking stage and passed a function pointer (usually the `mark_object()` function), which is used to mark an object. All Lisp objects that are contained within the object need to be marked by applying this function to them. The mark method should also return a Lisp object, which should be either nil or an object to mark. (This can be used in lieu of calling `mark_object()` on the object, to reduce the recursion depth, and consequently should be the most heavily nested sub-object, such as a long list.)

   Please note: When the mark method is called, garbage collection is in progress, and special precautions need to be taken when accessing objects; see section (B) above.

   If your mark method does not need to do anything, it can be `NULL`.

2. A *print* method. This is called to create a printed representation of the object, whenever `princ`, `prin1`, or the like is called. It is passed the object, a stream to which the output is to be directed, and an `escapeflag` which indicates whether the object's printed representation should be *escaped* so that it is readable. (This corresponds to the difference between `princ` and `prin1`.) Basically, *escaped* means that strings will have quotes around them and confusing characters in the strings such as quotes, backslashes, and newlines will be backslashed; and that special care will be taken to make symbols print in a readable fashion (e.g. symbols that look like numbers will be backslashed). Other readable objects should perhaps pass `escapeflag` on when sub-objects are printed, so that readability is preserved when necessary (or if not, always pass in a 1 for `escapeflag`). Non-readable objects should in general ignore `escapeflag`, except that some use it as an indication that more verbose output should be given.

   Sub-objects are printed using `print_internal()`, which takes exactly the same arguments as are passed to the print method.

Literal C strings should be printed using `write_c_string()`, or `write_string_1()` for non-null-terminated strings.

Functions that do not have a readable representation should check the `print_readably` flag and signal an error if it is set.

If you specify NULL for the print method, the `default_object_printer()` will be used.

3. A *finalize* method. This is called at the beginning of the sweep stage on lcrecords that are about to be freed, and should be used to perform any extra object cleanup. This typically involves freeing any extra `malloc()`ed memory associated with the object, releasing any operating-system and window-system resources associated with the object (e.g. pixmaps, fonts), etc.

   The finalize method can be NULL if nothing needs to be done.

   WARNING #1: The finalize method is also called at the end of the dump phase; this time with the for_disksave parameter set to non-zero. The object is *not* about to disappear, so you have to make sure to *not* free any extra `malloc()`ed memory if you're going to need it later. (Also, signal an error if there are any operating-system and window-system resources here, because they can't be dumped.)

   Finalize methods should, as a rule, set to zero any pointers after they've been freed, and check to make sure pointers are not zero before freeing. Although I'm pretty sure that finalize methods are not called twice on the same object (except for the `for_disksave` proviso), we've gotten nastily burned in some cases by not doing this.

   WARNING #2: The finalize method is *only* called for lcrecords, *not* for simply lrecords. If you need a finalize method for simple lrecords, you have to stick it in the `ADDITIONAL_FREE_foo()` macro in '`alloc.c`'.

   WARNING #3: Things are in an *extremely* bizarre state when `ADDITIONAL_FREE_foo()` is called, so you have to be incredibly careful when writing one of these functions. See the comment in `gc_sweep()`. If you ever have to add one of these, consider using an lcrecord or dealing with the problem in a different fashion.

4. An *equal* method. This compares the two objects for similarity, when `equal` is called. It should compare the contents of the objects in some reasonable fashion. It is passed the two objects and a *depth* value, which is used to catch circular objects. To compare sub-Lisp-objects, call `internal_equal()` and bump the depth value by one. If this value gets too high, a `circular-object` error will be signaled.

   If this is NULL, objects are `equal` only when they are `eq`, i.e. identical.

5. A *hash* method. This is used to hash objects when they are to be compared with `equal`. The rule here is that if two objects are `equal`, they *must* hash to the same value; i.e. your hash function should use some subset of the sub-fields of the object that are compared in the "equal" method. If you specify this method as `NULL`, the object's pointer will be used as the hash, which will *fail* if the object has an `equal` method, so don't do this.

   To hash a sub-Lisp-object, call `internal_hash()`. Bump the depth by one, just like in the "equal" method.

   To convert a Lisp object directly into a hash value (using its pointer), use `LISP_HASH()`. This is what happens when the hash method is NULL.

To hash two or more values together into a single value, use `HASH2()`, `HASH3()`, `HASH4()`, etc.

6. *getprop*, *putprop*, *remprop*, and *plist* methods. These are used for object types that have properties. I don't feel like documenting them here. If you create one of these objects, you have to use different macros to define them, i.e. `DEFINE_LRECORD_IMPLEMENTATION_WITH_PROPS()` or `DEFINE_LRECORD_SEQUENCE_IMPLEMENTATION_WITH_PROPS()`.
7. A *size_in_bytes* method, when the object is of variable-size. (i.e. declared with a `_SEQUENCE_IMPLEMENTATION` macro.) This should simply return the object's size in bytes, exactly as you might expect. For an example, see the methods for window configurations and opaques.

10.7 Low-level allocation

Memory that you want to allocate directly should be allocated using `xmalloc()` rather than `malloc()`. This implements error-checking on the return value, and once upon a time did some more vital stuff (i.e. `BLOCK_INPUT`, which is no longer necessary). Free using `xfree()`, and realloc using `xrealloc()`. Note that `xmalloc()` will do a non-local exit if the memory can't be allocated. (Many functions, however, do not expect this, and thus XEmacs will likely crash if this happens. **This is a bug.** If you can, you should strive to make your function handle this OK. However, it's difficult in the general circumstance, perhaps requiring extra unwind-protects and such.)

Note that XEmacs provides two separate replacements for the standard `malloc()` library function. These are called *old GNU malloc* ('`malloc.c`') and *new GNU malloc* ('`gmalloc.c`'), respectively. New GNU malloc is better in pretty much every way than old GNU malloc, and should be used if possible. (It used to be that on some systems, the old one worked but the new one didn't. I think this was due specifically to a bug in SunOS, which the new one now works around; so I don't think the old one ever has to be used any more.) The primary difference between both of these mallocs and the standard system malloc is that they are much faster, at the expense of increased space. The basic idea is that memory is allocated in fixed chunks of powers of two. This allows for basically constant malloc time, since the various chunks can just be kept on a number of free lists. (The standard system malloc typically allocates arbitrary-sized chunks and has to spend some time, sometimes a significant amount of time, walking the heap looking for a free block to use and cleaning things up.) The new GNU malloc improves on things by allocating large objects in chunks of 4096 bytes rather than in ever larger powers of two, which results in ever larger wastage. There is a slight speed loss here, but it's of doubtful significance.

NOTE: Apparently there is a third-generation GNU malloc that is significantly better than the new GNU malloc, and should probably be included in XEmacs.

There is also the relocating allocator, '`ralloc.c`'. This actually moves blocks of memory around so that the `sbrk()` pointer shrunk and virtual memory released back to the system. On some systems, this is a big win. On all systems, it causes a noticeable (and sometimes huge) speed penalty, so I turn it off by default. '`ralloc.c`' only works with the new GNU malloc in '`gmalloc.c`'. There are also two versions of '`ralloc.c`', one that uses `mmap()` rather than block copies to move data around. This purports to be faster, although that depends on the amount of data that would have had to be block copied and the system-call overhead for `mmap()`. I don't know exactly how this works, except that the relocating-allocation routines are pretty much used only for the memory allocated for a buffer, which is the biggest consumer of space, esp. of space that may get freed later.

Note that the GNU mallocs have some "memory warning" facilities. XEmacs taps into them and issues a warning through the standard warning system, when memory gets to 75%, 85%, and 95% full. (On some systems, the memory warnings are not functional.)

Allocated memory that is going to be used to make a Lisp object is created using `allocate_lisp_storage()`. This calls `xmalloc()` but also verifies that the pointer to the memory can fit into a Lisp word (remember that some bits are taken away for a type tag and a mark bit). If not, an error is issued through `memory_full()`. `allocate_lisp_storage()` is called by `alloc_lcrecord()`, `ALLOCATE_FIXED_TYPE()`, and the vector and bit-vector creation routines. These routines also call `INCREMENT_CONS_COUNTER()` at the appropriate times; this keeps statistics on how much memory is allocated, so that garbage-collection can be invoked when the threshold is reached.

10.8 Pure Space

Not yet documented.

10.9 Cons

Conses are allocated in standard frob blocks. The only thing to note is that conses can be explicitly freed using `free_cons()` and associated functions `free_list()` and `free_alist()`. This immediately puts the conses onto the cons free list, and decrements the statistics on memory allocation appropriately. This is used to good effect by some extremely commonly-used code, to avoid generating extra objects and thereby triggering GC sooner. However, you have to be *extremely* careful when doing this. If you mess this up, you will get BADLY BURNED, and it has happened before.

10.10 Vector

As mentioned above, each vector is `malloc()`ed individually, and all are threaded through the variable `all_vectors`. Vectors are marked strangely during garbage collection, by kludging the size field. Note that the `struct Lisp_Vector` is declared with its `contents` field being a *stretchy* array of one element. It is actually `malloc()`ed with the right size, however, and access to any element through the `contents` array works fine.

10.11 Bit Vector

Bit vectors work exactly like vectors, except for more complicated code to access an individual bit, and except for the fact that bit vectors are lrecords while vectors are not. (The only difference here is that there's an lrecord implementation pointer at the beginning and the tag field in bit vector Lisp words is "lrecord" rather than "vector".)

10.12 Symbol

Symbols are also allocated in frob blocks. Note that the code exists for symbols to be either lrecords (category (c) above) or simple types (category (b) above), and are lrecords by default (I think), although there is no good reason for this.

Note that symbols in the awful horrible obarray structure are chained through their `next` field.

Remember that `intern` looks up a symbol in an obarray, creating one if necessary.

10.13 Marker

Markers are allocated in frob blocks, as usual. They are kept in a buffer unordered, but in a doubly-linked list so that they can easily be removed. (Formerly this was a singly-linked list, but in some cases garbage collection took an extraordinarily long time due to the O(N^2) time required to remove lots of markers from a buffer.) Markers are removed from a buffer in the finalize stage, in `ADDITIONAL_FREE_marker()`.

10.14 String

As mentioned above, strings are a special case. A string is logically two parts, a fixed-size object (containing the length, property list, and a pointer to the actual data), and the actual data in the string. The fixed-size object is a `struct Lisp_String` and is allocated in frob blocks, as usual. The actual data is stored in special *string-chars blocks*, which are 8K blocks of memory. Currently-allocated strings are simply laid end to end in these string-chars blocks, with a pointer back to the `struct Lisp_String` stored before each string in the string-chars block. When a new string needs to be allocated, the remaining space at the end of the last string-chars block is used if there's enough, and a new string-chars block is created otherwise.

There are never any holes in the string-chars blocks due to the string compaction and relocation that happens at the end of garbage collection. During the sweep stage of garbage collection, when objects are reclaimed, the garbage collector goes through all string-chars blocks, looking for unused strings. Each chunk of string data is preceded by a pointer to the corresponding `struct Lisp_String`, which indicates both whether the string is used and how big the string is, i.e. how to get to the next chunk of string data. Holes are compressed by block-copying the next string into the empty space and relocating the pointer stored in the corresponding `struct Lisp_String`. **This means you have to be careful with strings in your code.** See the section above on `GCPRO`ing.

Note that there is one situation not handled: a string that is too big to fit into a string-chars block. Such strings, called *big strings*, are all `malloc()`ed as their own block. (#### Although it would make more sense for the threshold for big strings to be somewhat lower, e.g. 1/2 or 1/4 the size of a string-chars block. It seems that this was indeed the case formerly – indeed, the threshold was set at 1/8 – but Mly forgot about this when rewriting things for 19.8.)

Note also that the string data in string-chars blocks is padded as necessary so that proper alignment constraints on the `struct Lisp_String` back pointers are maintained.

Finally, strings can be resized. This happens in Mule when a character is substituted with a different-length character, or during modeline frobbing. (You could also export this to Lisp, but it's not done so currently.) Resizing a string is a potentially tricky process. If the change is small enough that the padding can absorb it, nothing other than a simple memory move needs to be done. Keep in mind, however, that the string can't shrink too much because the offset to the next string in the string-chars block is computed by looking at the length and rounding to the nearest multiple of four or eight. If the string would shrink or expand beyond the correct padding, new string data needs to be allocated at the end of the last string-chars block and the data moved appropriately. This leaves some dead string data, which is marked by putting a special marker of 0xFFFFFFFF in the `struct Lisp_String` pointer before the data (there's no real `struct Lisp_String` to point to and relocate), and storing the size of the dead string data (which would normally be obtained from the now-non-existent `struct Lisp_String`) at the beginning of the dead string data gap. The string compactor recognizes this special 0xFFFFFFFF marker and handles it correctly.

10.15 Bytecode

Not yet documented.

11 Events and the Event Loop

11.1 Introduction to Events

An event is an object that encapsulates information about an interesting occurrence in the operating system. Events are generated either by user action, direct (e.g. typing on the keyboard or moving the mouse) or indirect (moving another window, thereby generating an expose event on an Emacs frame), or as a result of some other typically asynchronous action happening, such as output from a subprocess being ready or a timer expiring. Events come into the system in an asynchronous fashion (typically through a callback being called) and are converted into a synchronous event queue (first-in, first-out) in a process that we will call *collection*.

Note that each application has its own event queue. (It is immaterial whether the collection process directly puts the events in the proper application's queue, or puts them into a single system queue, which is later split up.)

The most basic level of event collection is done by the operating system or window system. Typically, XEmacs does its own event collection as well. Often there are multiple layers of collection in XEmacs, with events from various sources being collected into a queue, which is then combined with other sources to go into another queue (i.e. a second level of collection), with perhaps another level on top of this, etc.

XEmacs has its own types of events (called *Emacs events*), which provides an abstract layer on top of the system-dependent nature of the most basic events that are received. Part of the complex nature of the XEmacs event collection process involves converting from the operating-system events into the proper Emacs events – there may not be a one-to-one correspondence.

Emacs events are documented in '`events.h`'; I'll discuss them later.

11.2 Main Loop

The *command loop* is the top-level loop that the editor is always running. It loops endlessly, calling `next-event` to retrieve an event and `dispatch-event` to execute it. `dispatch-event` does the appropriate thing with non-user events (process, timeout, magic, eval, mouse motion); this involves calling a Lisp handler function, redrawing a newly-exposed part of a frame, reading subprocess output, etc. For user events, `dispatch-event` looks up the event in relevant keymaps or menubars; when a full key sequence or menubar selection is reached, the appropriate function is executed. `dispatch-event` may have to keep state across calls; this is done in the "command-builder" structure associated with each console (remember, there's usually only one console), and the engine that looks up keystrokes and constructs full key sequences is called the *command builder*. This is documented elsewhere.

The guts of the command loop are in `command_loop_1()`. This function doesn't catch errors, though – that's the job of `command_loop_2()`, which is a condition-case (i.e. error-trapping) wrapper around `command_loop_1()`. `command_loop_1()` never returns, but may get thrown out of.

When an error occurs, `cmd_error()` is called, which usually invokes the Lisp error handler in `command-error`; however, a default error handler is provided if `command-error` is `nil` (e.g. during startup). The purpose of the error handler is simply to display the error message and do associated cleanup; it does not need to throw anywhere. When the error handler finishes, the condition-case in `command_loop_2()` will finish and `command_loop_2()` will reinvoke `command_loop_1()`.

`command_loop_2()` is invoked from three places: from `initial_command_loop()` (called from `main()` at the end of internal initialization), from the Lisp function `recursive-edit`, and from `call_command_loop()`.

`call_command_loop()` is called when a macro is started and when the minibuffer is entered; normal termination of the macro or minibuffer causes a throw out of the recursive command loop. (To `execute-kbd-macro` for macros and `exit` for minibuffers. Note also that the low-level minibuffer-entering function, `read-minibuffer-internal`, provides its own error handling and does not need `command_loop_2()`'s error encapsulation; so it tells `call_command_loop()` to invoke `command_loop_1()` directly.)

Note that both read-minibuffer-internal and recursive-edit set up a catch for `exit`; this is why `abort-recursive-edit`, which throws to this catch, exits out of either one.

`initial_command_loop()`, called from `main()`, sets up a catch for `top-level` when invoking `command_loop_2()`, allowing functions to throw all the way to the top level if they really need to. Before invoking `command_loop_2()`, `initial_command_loop()` calls `top_level_1()`, which handles all of the startup stuff (creating the initial frame, handling the command-line options, loading the user's '`.emacs`' file, etc.). The function that actually does this is in Lisp and is pointed to by the variable `top-level`; normally this function is `normal-top-level`. `top_level_1()` is just an error-handling wrapper similar to `command_loop_2()`. Note also that `initial_command_loop()` sets up a catch for `top-level` when invoking `top_level_1()`, just like when it invokes `command_loop_2()`.

11.3 Specifics of the Event Gathering Mechanism

Here is an approximate diagram of the collection processes at work in XEmacs, under TTY's (TTY's are simpler than X so we'll look at this first):

```
     asynch.      asynch.    asynch.   asynch.             [Collectors in
   kbd events  kbd events   process   process                 the OS]
         |          |        output    output
         |          |          |         |
         |          |          |         |     SIGINT,      [signal handlers
         |          |          |         |     SIGQUIT,        in XEmacs]
         V          V          V         V     SIGWINCH,
        file       file       file      file   SIGALRM
        desc.      desc.      desc.     desc.    |
        (TTY)      (TTY)      (pipe)    (pipe)   |
         |           |          |         |     fake    timeouts
         |           |          |         |     file        |
         |           |          |         |     desc.       |
         |           |          |         |     (pipe)      |
         |           |          |         |       |         |
```

```
       |          |          |         |        |         |
       |          |          |         |        |         |
       V          V          V         V        V         V
       ------>-----------<----------------<----------------
                    |
                    |
                    |   [collected using select() in emacs_tty_next_event()
                    |    and converted to the appropriate Emacs event]
                    |
                    |
                    V              (above this line is TTY-specific)
                  Emacs    ------------------------------------------------
                  event    (below this line is the generic event mechanism)
                    |
                    |
 was there      if not, call
 a SIGINT?   emacs_tty_next_event()
     |              |
     |              |
     |              |
     V              V
     --->-------<----
            |
            |        [collected in event_stream_next_event();
            |         SIGINT is converted using maybe_read_quit_event()]
            V
          Emacs
          event
            |
            \---->------>----- maybe_kbd_translate() ---->---\
                                                             |
                                                             |
                                                             |
      command event queue                                    |
                                                  if not from command
   (contains events that were                     event queue, call
   read earlier but not processed,                event_stream_next_event()
   typically when waiting in a                               |
   sit-for, sleep-for, etc. for                              |
  a particular event to be received)                         |
                |                                            |
                |                                            |
                V                                            V
                ---->------------------------------------<----
                                                |
                                                |   [collected in
                                                |    next_event_internal()]
                                                |
  unread-     unread-       event from          |
```

```
command-    command-       keyboard       else, call
events      event           macro     next_event_internal()
  |           |               |               |
  |           |               |               |
  |           |               |               |
  V           V               V               V
  --------->---------------------<------------
                  |
                  |      [collected in `next-event', which may loop
                  |       more than once if the event it gets is on
                  |       a dead frame, device, etc.]
                  |
                  |
                  V
         feed into top-level event loop,
         which repeatedly calls `next-event'
         and then dispatches the event
         using `dispatch-event'
```

Notice the separation between TTY-specific and generic event mechanism. When using the Xt-based event loop, the TTY-specific stuff is replaced but the rest stays the same.

It's also important to realize that only one different kind of system-specific event loop can be operating at a time, and must be able to receive all kinds of events simultaneously. For the two existing event loops (implemented in '`event-tty.c`' and '`event-Xt.c`', respectively), the TTY event loop *only* handles TTY consoles, while the Xt event loop handles *both* TTY and X consoles. This situation is different from all of the output handlers, where you simply have one per console type.

Here's the Xt Event Loop Diagram (notice that below a certain point, it's the same as the above diagram):

```
asynch. asynch. asynch. asynch.                 [Collectors in
 kbd     kbd    process process                    the OS]
events  events  output  output
  |       |       |       |
  |       |       |       |     asynch. asynch.   [Collectors in the
  |       |       |       |       X       X        OS and X Window System]
  |       |       |       |     events  events
  |       |       |       |       |       |
  |       |       |       |       |       |
  |       |       |       |       |       |    SIGINT,   [signal handlers
  |       |       |       |       |       |    SIGQUIT,     in XEmacs]
  |       |       |       |       |       |    SIGWINCH,
  |       |       |       |       |       |    SIGALRM
  |       |       |       |       |       |       |
  |       |       |       |       |       |       |
  |       |       |       |       |       |       |      timeouts
  |       |       |       |       |       |       |          |
  |       |       |       |       |       |       |          |
  |       |       |       |       |       |       V          |
  V       V       V       V       V       V      fake        |
```

```
file    file    file    file    file     file   file            |
desc.   desc.   desc.   desc.   desc.    desc.  desc.           |
(TTY)   (TTY)   (pipe)  (pipe) (socket) (socket) (pipe)         |
 |       |       |       |       |        |       |          |
 |       |       |       |       |        |       |          |
 |       |       |       |       |        |       |          |
 V       V       V       V       V        V       V          V
 --->----------------------------------------<---------<------
      |              |               |
      |              |               |   [collected using select() in
      |              |               |   _XtWaitForSomething(), called
      |              |               |   from XtAppProcessEvent(), called
      |              |               |   in emacs_Xt_next_event();
      |              |               |   dispatched to various callbacks]
      |              |               |
      |              |               |
 emacs_Xt_        p_s_callback(),    |   [popup_selection_callback]
 event_handler()  x_u_v_s_callback(),|   [x_update_vertical_scrollbar_
      |           x_u_h_s_callback(),|    callback]
      |           search_callback()  |   [x_update_horizontal_scrollbar_
      |              |               |    callback]
      |              |               |
      |              |               |
 enqueue_Xt_       signal_special_   |
 dispatch_event()  Xt_user_event()   |
 [maybe multiple     |               |
  times, maybe 0     |               |
  times]             |               |
      |            enqueue_Xt_       |
      |            dispatch_event()  |
      |              |               |
      |              |               |
      V              V               |
      -->----------<--               |
             |                       |
             |                       |
          dispatch              Xt_what_callback()
          event                   sets flags
          queue                      |
             |                       |
             |                       |
             |                       |
             |                       |
             ---->-----------<--------
                  |
                  |
                  |     [collected and converted as appropriate in
                  |            emacs_Xt_next_event()]
                  |
```

```
                   |
                   V              (above this line is Xt-specific)
                 Emacs    ------------------------------------------------
                 event    (below this line is the generic event mechanism)
                   |
                   |
was there      if not, call
a SIGINT?   emacs_Xt_next_event()
    |              |
    |              |
    |              |
    V              V
    --->-------<----
           |
           |        [collected in event_stream_next_event();
           |         SIGINT is converted using maybe_read_quit_event()]
           V
         Emacs
         event
           |
           \---->------>----- maybe_kbd_translate() -->-----\
                                                            |
                                                            |
                                                            |
     command event queue                                    |
                                                 if not from command
  (contains events that were                     event queue, call
  read earlier but not processed,                event_stream_next_event()
  typically when waiting in a                               |
  sit-for, sleep-for, etc. for                              |
 a particular event to be received)                         |
               |                                            |
               |                                            |
               V                                            V
               ---->----------------------------------<------
                                               |
                                               |   [collected in
                                               |    next_event_internal()]
                                               |
 unread-     unread-       event from          |
 command-    command-       keyboard       else, call
 events      event           macro      next_event_internal()
   |           |               |               |
   |           |               |               |
   |           |               |               |
   V           V               V               V
   --------->----------------------<------------
                     |
                     |      [collected in ‘next-event’, which may loop
```

```
                  |       more than once if the event it gets is on
                  |       a dead frame, device, etc.]
                  |
                  |
                  V
         feed into top-level event loop,
         which repeatedly calls 'next-event'
         and then dispatches the event
         using 'dispatch-event'
```

11.4 Specifics About the Emacs Event

11.5 The Event Stream Callback Routines

11.6 Other Event Loop Functions

`detect_input_pending()` and `input-pending-p` look for input by calling `event_stream->event_pending_p` and looking in `[V]unread-command-event` and the `command_event_queue` (they do not check for an executing keyboard macro, though).

`discard-input` cancels any command events pending (and any keyboard macros currently executing), and puts the others onto the `command_event_queue`. There is a comment about a "race condition", which is not a good sign.

`next-command-event` and `read-char` are higher-level interfaces to `next-event`. `next-command-event` gets the next *command* event (i.e. keypress, mouse event, menu selection, or scrollbar action), calling `dispatch-event` on any others. `read-char` calls `next-command-event` and uses `event_to_character()` to return the character equivalent. With the right kind of input method support, it is possible for (read-char) to return a Kanji character.

11.7 Converting Events

`character_to_event()`, `event_to_character()`, `event-to-character`, and `character-to-event` convert between characters and keypress events corresponding to the characters. If the event was not a keypress, `event_to_character()` returns -1 and `event-to-character` returns `nil`. These functions convert between character representation and the split-up event representation (keysym plus mod keys).

11.8 Dispatching Events; The Command Builder

Not yet documented.

12 Evaluation; Stack Frames; Bindings

12.1 Evaluation

`Feval()` evaluates the form (a Lisp object) that is passed to it. Note that evaluation is only non-trivial for two types of objects: symbols and conses. A symbol is evaluated simply by calling symbol-value on it and returning the value.

Evaluating a cons means calling a function. First, `eval` checks to see if garbage-collection is necessary, and calls `Fgarbage_collect()` if so. It then increases the evaluation depth by 1 (`lisp_eval_depth`, which is always less than `max_lisp_eval_depth`) and adds an element to the linked list of `struct backtrace`'s (`backtrace_list`). Each such structure contains a pointer to the function being called plus a list of the function's arguments. Originally these values are stored unevalled, and as they are evaluated, the backtrace structure is updated. Garbage collection pays attention to the objects pointed to in the backtrace structures (garbage collection might happen while a function is being called or while an argument is being evaluated, and there could easily be no other references to the arguments in the argument list; once an argument is evaluated, however, the unevalled version is not needed by eval, and so the backtrace structure is changed).

At this point, the function to be called is determined by looking at the car of the cons (if this is a symbol, its function definition is retrieved and the process repeated). The function should then consist of either a `Lisp_Subr` (built-in function), a `Lisp_Compiled_Function` object, or a cons whose car is the symbol `autoload`, `macro` or `lambda`.

If the function is a `Lisp_Subr`, the lisp object points to a `struct Lisp_Subr` (created by `DEFUN()`), which contains a pointer to the C function, a minimum and maximum number of arguments (possibly the special constants `MANY` or `UNEVALLED`), a pointer to the symbol referring to that subr, and a couple of other things. If the subr wants its arguments `UNEVALLED`, they are passed raw as a list. Otherwise, an array of evaluated arguments is created and put into the backtrace structure, and either passed whole (`MANY`) or each argument is passed as a C argument.

If the function is a `Lisp_Compiled_Function` object or a lambda, `apply_lambda()` is called. If the function is a macro, [..... fill in] is done. If the function is an autoload, `do_autoload()` is called to load the definition and then eval starts over [explain this more].

When `Feval` exits, the evaluation depth is reduced by one, the debugger is called if appropriate, and the current backtrace structure is removed from the list.

`apply_lambda()` is passed a function, a list of arguments, and a flag indicating whether to evaluate the arguments. It creates an array of (possibly) evaluated arguments and fixes up the backtrace structure, just like eval does. Then it calls `funcall_lambda()`.

`funcall_lambda()` goes through the formal arguments to the function and binds them to the actual arguments, checking for `&rest` and `&optional` symbols in the formal arguments and making sure the number of actual arguments is correct. Then either `progn` or `byte-code` is called to actually execute the body and return a value.

`Ffuncall()` implements Lisp `funcall`. `(funcall fun x1 x2 x3 ...)` is equivalent to `(eval (list fun (quote x1) (quote x2) (quote x3) ...))`. `Ffuncall()` contains its own code to do the evaluation, however, and is almost identical to eval.

`Fapply()` implements Lisp `apply`, which is very similar to `funcall` except that if the last argument is a list, the result is the same as if each of the arguments in the list had been passed separately. `Fapply()` does some business to expand the last argument if it's a list, then calls `Ffuncall()` to do the work.

`apply1()`, `call0()`, `call1()`, `call2()`, and `call3()` call a function, passing it the argument(s) given (the arguments are given as separate C arguments rather than being passed as an array). `apply1()` uses `apply` while the others use `funcall`.

12.2 Dynamic Binding; The specbinding Stack; Unwind-Protects

```
struct specbinding
{
  Lisp_Object symbol, old_value;
  Lisp_Object (*func) (Lisp_Object); /* for unwind-protect */
};
```

`struct specbinding` is used for local-variable bindings and unwind-protects. `specpdl` holds an array of `struct specbinding`'s, `specpdl_ptr` points to the beginning of the free bindings in the array, `specpdl_size` specifies the total number of binding slots in the array, and `max_specpdl_size` specifies the maximum number of bindings the array can be expanded to hold. `grow_specpdl()` increases the size of the `specpdl` array, multiplying its size by 2 but never exceeding `max_specpdl_size` (except that if this number is less than 400, it is first set to 400).

`specbind()` binds a symbol to a value and is used for local variables and `let` forms. The symbol and its old value (which might be `Qunbound`, indicating no prior value) are recorded in the specpdl array, and `specpdl_size` is increased by 1.

`record_unwind_protect()` implements an *unwind-protect*, which, when placed around a section of code, ensures that some specified cleanup routine will be executed even if the code exits abnormally (e.g. through a `throw` or quit). `record_unwind_protect()` simply adds a new specbinding to the `specpdl` array and stores the appropriate information in it. The cleanup routine can either be a C function, which is stored in the `func` field, or a `progn` form, which is stored in the `old_value` field.

`unbind_to()` removes specbindings from the `specpdl` array until the specified position is reached. Each specbinding can be one of three types:

1. an unwind-protect with a C cleanup function (`func` is not 0, and `old_value` holds an argument to be passed to the function);
2. an unwind-protect with a Lisp form (`func` is 0, `symbol` is `nil`, and `old_value` holds the form to be executed with `Fprogn()`); or
3. a local-variable binding (`func` is 0, `symbol` is not `nil`, and `old_value` holds the old value, which is stored as the symbol's value).

12.3 Simple Special Forms

`or`, `and`, `if`, `cond`, `progn`, `prog1`, `prog2`, `setq`, `quote`, `function`, `let*`, `let`, `while`

All of these are very simple and work as expected, calling `Feval()` or `Fprogn()` as necessary and (in the case of `let` and `let*`) using `specbind()` to create bindings and `unbind_to()` to undo the bindings when finished. Note that these functions do a lot of `GCPRO`ing to protect their arguments from garbage collection because they call `Feval()` (see Section 10.2 [Garbage Collection], page 62).

12.4 Catch and Throw

```
struct catchtag
{
  Lisp_Object tag;
  Lisp_Object val;
  struct catchtag *next;
  struct gcpro *gcpro;
  jmp_buf jmp;
  struct backtrace *backlist;
  int lisp_eval_depth;
  int pdlcount;
};
```

`catch` is a Lisp function that places a catch around a body of code. A catch is a means of non-local exit from the code. When a catch is created, a tag is specified, and executing a `throw` to this tag will exit from the body of code caught with this tag, and its value will be the value given in the call to `throw`. If there is no such call, the code will be executed normally.

Information pertaining to a catch is held in a `struct catchtag`, which is placed at the head of a linked list pointed to by `catchlist`. `internal_catch()` is passed a C function to call (`Fprogn()` when Lisp `catch` is called) and arguments to give it, and places a catch around the function. Each `struct catchtag` is held in the stack frame of the `internal_catch()` instance that created the catch.

`internal_catch()` is fairly straightforward. It stores into the `struct catchtag` the tag name and the current values of `backtrace_list`, `lisp_eval_depth`, `gcprolist`, and the offset into the `specpdl` array, sets a jump point with `_setjmp()` (storing the jump point into the `struct catchtag`), and calls the function. Control will return to `internal_catch()` either when the function exits normally or through a `_longjmp()` to this jump point. In the latter case, `throw` will store the value to be returned into the `struct catchtag` before jumping. When it's done, `internal_catch()` removes the `struct catchtag` from the catchlist and returns the proper value.

`Fthrow()` goes up through the catchlist until it finds one with a matching tag. It then calls `unbind_catch()` to restore everything to what it was when the appropriate catch was set, stores the return value in the `struct catchtag`, and jumps (with `_longjmp()`) to its jump point.

`unbind_catch()` removes all catches from the catchlist until it finds the correct one. Some of the catches might have been placed for error-trapping, and if so, the appropriate entries on the handlerlist must be removed (see "errors"). `unbind_catch()` also restores the values of `gcprolist`, `backtrace_list`, and `lisp_eval`, and calls `unbind_to()` to undo any specbindings created since the catch.

13 Symbols and Variables

13.1 Introduction to Symbols

A symbol is basically just an object with four fields: a name (a string), a value (some Lisp object), a function (some Lisp object), and a property list (usually a list of alternating keyword/value pairs). What makes symbols special is that there is usually only one symbol with a given name, and the symbol is referred to by name. This makes a symbol a convenient way of calling up data by name, i.e. of implementing variables. (The variable's value is stored in the *value slot*.) Similarly, functions are referenced by name, and the definition of the function is stored in a symbol's *function slot*. This means that there can be a distinct function and variable with the same name. The property list is used as a more general mechanism of associating additional values with particular names, and once again the namespace is independent of the function and variable namespaces.

13.2 Obarrays

The identity of symbols with their names is accomplished through a structure called an obarray, which is just a poorly-implemented hash table mapping from strings to symbols whose name is that string. (I say "poorly implemented" because an obarray appears in Lisp as a vector with some hidden fields rather than as its own opaque type. This is an Emacs Lisp artifact that should be fixed.)

Obarrays are implemented as a vector of some fixed size (which should be a prime for best results), where each "bucket" of the vector contains one or more symbols, threaded through a hidden `next` field in the symbol. Lookup of a symbol in an obarray, and adding a symbol to an obarray, is accomplished through standard hash-table techniques.

The standard Lisp function for working with symbols and obarrays is `intern`. This looks up a symbol in an obarray given its name; if it's not found, a new symbol is automatically created with the specified name, added to the obarray, and returned. This is what happens when the Lisp reader encounters a symbol (or more precisely, encounters the name of a symbol) in some text that it is reading. There is a standard obarray called `obarray` that is used for this purpose, although the Lisp programmer is free to create his own obarrays and `intern` symbols in them.

Note that, once a symbol is in an obarray, it stays there until something is done about it, and the standard obarray `obarray` always stays around, so once you use any particular variable name, a corresponding symbol will stay around in `obarray` until you exit XEmacs.

Note that `obarray` itself is a variable, and as such there is a symbol in `obarray` whose name is `"obarray"` and which contains `obarray` as its value.

Note also that this call to `intern` occurs only when in the Lisp reader, not when the code is executed (at which point the symbol is already around, stored as such in the definition of the function).

You can create your own obarray using `make-vector` (this is horrible but is an artifact) and intern symbols into that obarray. Doing that will result in two or more symbols with the same name. However, at most one of these symbols is in the standard `obarray`: You cannot have two symbols of the same name in any particular obarray. Note that you cannot add a symbol to an obarray in any fashion other than using `intern`: i.e. you can't take an existing symbol and put it in an existing obarray. Nor can you change the name of an existing symbol. (Since obarrays are vectors, you can violate the consistency of things by storing directly into the vector, but let's ignore that possibility.)

Usually symbols are created by `intern`, but if you really want, you can explicitly create a symbol using `make-symbol`, giving it some name. The resulting symbol is not in any obarray (i.e. it is *uninterned*), and you can't add it to any obarray. Therefore its primary purpose is as a symbol to use in macros to avoid namespace pollution. It can also be used as a carrier of information, but cons cells could probably be used just as well.

You can also use `intern-soft` to look up a symbol but not create a new one, and `unintern` to remove a symbol from an obarray. This returns the removed symbol. (Remember: You can't put the symbol back into any obarray.) Finally, `mapatoms` maps over all of the symbols in an obarray.

13.3 Symbol Values

The value field of a symbol normally contains a Lisp object. However, a symbol can be *unbound*, meaning that it logically has no value. This is internally indicated by storing a special Lisp object, called *the unbound marker* and stored in the global variable `Qunbound`. The unbound marker is of a special Lisp object type called *symbol-value-magic*. It is impossible for the Lisp programmer to directly create or access any object of this type.

You must not let any "symbol-value-magic" object escape to the Lisp level. Printing any of these objects will cause the message '`INTERNAL EMACS BUG`' to appear as part of the print representation. (You may see this normally when you call `debug_print()` from the debugger on a Lisp object.) If you let one of these objects escape to the Lisp level, you will violate a number of assumptions contained in the C code and make the unbound marker not function right.

When a symbol is created, its value field (and function field) are set to `Qunbound`. The Lisp programmer can restore these conditions later using `makunbound` or `fmakunbound`, and can query to see whether the value of function fields are *bound* (i.e. have a value other than `Qunbound`) using `boundp` and `fboundp`. The fields are set to a normal Lisp object using `set` (or `setq`) and `fset`.

Other symbol-value-magic objects are used as special markers to indicate variables that have non-normal properties. This includes any variables that are tied into C variables (setting the variable magically sets some global variable in the C code, and likewise for retrieving the variable's value), variables that magically tie into slots in the current buffer, variables that are buffer-local, etc. The symbol-value-magic object is stored in the value cell in place of a normal object, and the code to retrieve a symbol's value (i.e. `symbol-value`) knows how to do special things with them. This means that you should not just fetch the value cell directly if you want a symbol's value.

The exact workings of this are rather complex and involved and are well-documented in comments in '`buffer.c`', '`symbols.c`', and '`lisp.h`'.

14 Buffers and Textual Representation

14.1 Introduction to Buffers

A buffer is logically just a Lisp object that holds some text. In this, it is like a string, but a buffer is optimized for frequent insertion and deletion, while a string is not. Furthermore:

1. Buffers are *permanent* objects, i.e. once you create them, they remain around, and need to be explicitly deleted before they go away.
2. Each buffer has a unique name, which is a string. Buffers are normally referred to by name. In this respect, they are like symbols.
3. Buffers have a default insertion position, called *point*. Inserting text (unless you explicitly give a position) goes at point, and moves point forward past the text. This is what is going on when you type text into Emacs.
4. Buffers have lots of extra properties associated with them.
5. Buffers can be *displayed*. What this means is that there exist a number of *windows*, which are objects that correspond to some visible section of your display, and each window has an associated buffer, and the current contents of the buffer are shown in that section of the display. The redisplay mechanism (which takes care of doing this) knows how to look at the text of a buffer and come up with some reasonable way of displaying this. Many of the properties of a buffer control how the buffer's text is displayed.
6. One buffer is distinguished and called the *current buffer*. It is stored in the variable `current_buffer`. Buffer operations operate on this buffer by default. When you are typing text into a buffer, the buffer you are typing into is always `current_buffer`. Switching to a different window changes the current buffer. Note that Lisp code can temporarily change the current buffer using `set-buffer` (often enclosed in a `save-excursion` so that the former current buffer gets restored when the code is finished). However, calling `set-buffer` will NOT cause a permanent change in the current buffer. The reason for this is that the top-level event loop sets `current_buffer` to the buffer of the selected window, each time it finishes executing a user command.

Make sure you understand the distinction between *current buffer* and *buffer of the selected window*, and the distinction between *point* of the current buffer and *window-point* of the selected window. (This latter distinction is explained in detail in the section on windows.)

14.2 The Text in a Buffer

The text in a buffer consists of a sequence of zero or more characters. A *character* is an integer that logically represents a letter, number, space, or other unit of text. Most of the characters that you will typically encounter belong to the ASCII set of characters, but there are also characters for various sorts of accented letters, special symbols, Chinese and Japanese ideograms (i.e. Kanji, Katakana, etc.), Cyrillic and Greek letters, etc. The actual number of possible characters is quite large.

For now, we can view a character as some non-negative integer that has some shape that defines how it typically appears (e.g. as an uppercase A). (The exact way in which a character appears depends on the font used to display the character.) The internal type of characters in the C code is an `Emchar`; this is just an `int`, but using a symbolic type makes the code clearer.

Between every character in a buffer is a *buffer position* or *character position*. We can speak of the character before or after a particular buffer position, and when you insert a character at a particular position, all characters after that position end up at new positions. When we speak of the character *at* a position, we really mean the character after the position. (This schizophrenia between a buffer position being "between" a character and "on" a character is rampant in Emacs.)

Buffer positions are numbered starting at 1. This means that position 1 is before the first character, and position 0 is not valid. If there are N characters in a buffer, then buffer position N+1 is after the last one, and position N+2 is not valid.

The internal makeup of the Emchar integer varies depending on whether we have compiled with MULE support. If not, the Emchar integer is an 8-bit integer with possible values from 0 - 255. 0 - 127 are the standard ASCII characters, while 128 - 255 are the characters from the ISO-8859-1 character set. If we have compiled with MULE support, an Emchar is a 19-bit integer, with the various bits having meanings according to a complex scheme that will be detailed later. The characters numbered 0 - 255 still have the same meanings as for the non-MULE case, though.

Internally, the text in a buffer is represented in a fairly simple fashion: as a contiguous array of bytes, with a *gap* of some size in the middle. Although the gap is of some substantial size in bytes, there is no text contained within it: From the perspective of the text in the buffer, it does not exist. The gap logically sits at some buffer position, between two characters (or possibly at the beginning or end of the buffer). Insertion of text in a buffer at a particular position is always accomplished by first moving the gap to that position (i.e. through some block moving of text), then writing the text into the beginning of the gap, thereby shrinking the gap. If the gap shrinks down to nothing, a new gap is created. (What actually happens is that a new gap is "created" at the end of the buffer's text, which requires nothing more than changing a couple of indices; then the gap is "moved" to the position where the insertion needs to take place by moving up in memory all the text after that position.) Similarly, deletion occurs by moving the gap to the place where the text is to be deleted, and then simply expanding the gap to include the deleted text. (*Expanding* and *shrinking* the gap as just described means just that the internal indices that keep track of where the gap is located are changed.)

Note that the total amount of memory allocated for a buffer text never decreases while the buffer is live. Therefore, if you load up a 20-megabyte file and then delete all but one character, there will be a 20-megabyte gap, which won't get any smaller (except by inserting characters back again). Once the buffer is killed, the memory allocated for the buffer text will be freed, but it will still be sitting on the heap, taking up virtual memory, and will not be released back to the operating system. (However, if you have compiled XEmacs with rel-alloc, the situation is different. In this case, the space *will* be released back to the operating system. However, this tends to result in a noticeable speed penalty.)

Astute readers may notice that the text in a buffer is represented as an array of *bytes*, while (at least in the MULE case) an Emchar is a 19-bit integer, which clearly cannot fit in a byte. This means (of course) that the text in a buffer uses a different representation from an Emchar: specifically, the 19-bit Emchar becomes a series of one to four bytes. The conversion between these two representations is complex and will be described later.

In the non-MULE case, everything is very simple: An Emchar is an 8-bit value, which fits neatly into one byte.

If we are given a buffer position and want to retrieve the character at that position, we need to follow these steps:

1. Pretend there's no gap, and convert the buffer position into a *byte index* that indexes to the appropriate byte in the buffer's stream of textual bytes. By convention, byte indices begin at 1, just like buffer positions. In the non-MULE case, byte indices and buffer positions are identical, since one character equals one byte.
2. Convert the byte index into a *memory index*, which takes the gap into account. The memory index is a direct index into the block of memory that stores the text of a buffer. This basically just involves checking to see if the byte index is past the gap, and if so, adding the size of the gap to it. By convention, memory indices begin at 1, just like buffer positions and byte indices, and when referring to the position that is *at* the gap, we always use the memory position at the *beginning*, not at the end, of the gap.
3. Fetch the appropriate bytes at the determined memory position.
4. Convert these bytes into an Emchar.

In the non-Mule case, (3) and (4) boil down to a simple one-byte memory access.

Note that we have defined three types of positions in a buffer:

1. *buffer positions* or *character positions*, typedef `Bufpos`
2. *byte indices*, typedef `Bytind`
3. *memory indices*, typedef `Memind`

All three typedefs are just `int`s, but defining them this way makes things a lot clearer.

Most code works with buffer positions. In particular, all Lisp code that refers to text in a buffer uses buffer positions. Lisp code does not know that byte indices or memory indices exist.

Finally, we have a typedef for the bytes in a buffer. This is a `Bufbyte`, which is an unsigned char. Referring to them as Bufbytes underscores the fact that we are working with a string of bytes in the internal Emacs buffer representation rather than in one of a number of possible alternative representations (e.g. EUC-encoded text, etc.).

14.3 Buffer Lists

Recall earlier that buffers are *permanent* objects, i.e. that they remain around until explicitly deleted. This entails that there is a list of all the buffers in existence. This list is actually an assoc-list (mapping from the buffer's name to the buffer) and is stored in the global variable `Vbuffer_alist`.

The order of the buffers in the list is important: the buffers are ordered approximately from most-recently-used to least-recently-used. Switching to a buffer using `switch-to-buffer`, `pop-to-buffer`, etc. and switching windows using `other-window`, etc. usually brings the new current buffer to the front of the list. `switch-to-buffer`, `other-buffer`, etc. look at the beginning of the list to find an alternative buffer to suggest. You can also explicitly move a buffer to the end of the list using `bury-buffer`.

In addition to the global ordering in `Vbuffer_alist`, each frame has its own ordering of the list. These lists always contain the same elements as in `Vbuffer_alist` although possibly in a different order. `buffer-list` normally returns the list for the selected frame. This allows you to work in separate frames without things interfering with each other.

The standard way to look up a buffer given a name is `get-buffer`, and the standard way to create a new buffer is `get-buffer-create`, which looks up a buffer with a given name, creating a new one if necessary. These operations correspond exactly with the symbol operations `intern-soft` and `intern`, respectively. You can also force a new buffer to be created using `generate-new-buffer`, which takes a name and (if necessary) makes a unique name from this by appending a number, and then creates the buffer. This is basically like the symbol operation `gensym`.

14.4 Markers and Extents

Among the things associated with a buffer are things that are logically attached to certain buffer positions. This can be used to keep track of a buffer position when text is inserted and deleted, so that it remains at the same spot relative to the text around it; to assign properties to particular sections of text; etc. There are two such objects that are useful in this regard: they are *markers* and *extents*.

A *marker* is simply a flag placed at a particular buffer position, which is moved around as text is inserted and deleted. Markers are used for all sorts of purposes, such as the `mark` that is the other end of textual regions to be cut, copied, etc.

An *extent* is similar to two markers plus some associated properties, and is used to keep track of regions in a buffer as text is inserted and deleted, and to add properties (e.g. fonts) to particular regions of text. The external interface of extents is explained elsewhere.

The important thing here is that markers and extents simply contain buffer positions in them as integers, and every time text is inserted or deleted, these positions must be updated. In order to minimize the amount of shuffling that needs to be done, the positions in markers and extents (there's one per marker, two per extent) and stored in Meminds. This means that they only need to be moved when the text is physically moved in memory; since the gap structure tries to minimize this, it also minimizes the number of marker and extent indices that need to be adjusted. Look in '`insdel.c`' for the details of how this works.

One other important distinction is that markers are *temporary* while extents are *permanent*. This means that markers disappear as soon as there are no more pointers to them, and correspondingly, there is no way to determine what markers are in a buffer if you are just given the buffer. Extents remain in a buffer until they are detached (which could happen as a result of text being deleted) or the buffer is deleted, and primitives do exist to enumerate the extents in a buffer.

14.5 Bufbytes and Emchars

Not yet documented.

14.6 The Buffer Object

Buffers contain fields not directly accessible by the Lisp programmer. We describe them here, naming them by the names used in the C code. Many are accessible indirectly in Lisp programs via Lisp primitives.

`name` The buffer name is a string that names the buffer. It is guaranteed to be unique. See section "Buffer Names" in *XEmacs Lisp Programmer's Manual*.

`save_modified`
: This field contains the time when the buffer was last saved, as an integer. See section "Buffer Modification" in *XEmacs Lisp Programmer's Manual*.

`modtime` This field contains the modification time of the visited file. It is set when the file is written or read. Every time the buffer is written to the file, this field is compared to the modification time of the file. See section "Buffer Modification" in *XEmacs Lisp Programmer's Manual*.

`auto_save_modified`
: This field contains the time when the buffer was last auto-saved.

`last_window_start`
: This field contains the `window-start` position in the buffer as of the last time the buffer was displayed in a window.

`undo_list`
: This field points to the buffer's undo list. See section "Undo" in *XEmacs Lisp Programmer's Manual*.

`syntax_table_v`
: This field contains the syntax table for the buffer. See section "Syntax Tables" in *XEmacs Lisp Programmer's Manual*.

`downcase_table`
: This field contains the conversion table for converting text to lower case. See section "Case Tables" in *XEmacs Lisp Programmer's Manual*.

`upcase_table`
: This field contains the conversion table for converting text to upper case. See section "Case Tables" in *XEmacs Lisp Programmer's Manual*.

`case_canon_table`
: This field contains the conversion table for canonicalizing text for case-folding search. See section "Case Tables" in *XEmacs Lisp Programmer's Manual*.

`case_eqv_table`
: This field contains the equivalence table for case-folding search. See section "Case Tables" in *XEmacs Lisp Programmer's Manual*.

`display_table`
: This field contains the buffer's display table, or `nil` if it doesn't have one. See section "Display Tables" in *XEmacs Lisp Programmer's Manual*.

`markers` This field contains the chain of all markers that currently point into the buffer. Deletion of text in the buffer, and motion of the buffer's gap, must check each of these markers and perhaps update it. See section "Markers" in *XEmacs Lisp Programmer's Manual*.

`backed_up`
: This field is a flag that tells whether a backup file has been made for the visited file of this buffer.

`mark`
: This field contains the mark for the buffer. The mark is a marker, hence it is also included on the list `markers`. See section "The Mark" in *XEmacs Lisp Programmer's Manual*.

`mark_active`
: This field is non-`nil` if the buffer's mark is active.

`local_var_alist`
: This field contains the association list describing the variables local in this buffer, and their values, with the exception of local variables that have special slots in the buffer object. (Those slots are omitted from this table.) See section "Buffer-Local Variables" in *XEmacs Lisp Programmer's Manual*.

`modeline_format`
: This field contains a Lisp object which controls how to display the mode line for this buffer. See section "Modeline Format" in *XEmacs Lisp Programmer's Manual*.

`base_buffer`
: This field holds the buffer's base buffer (if it is an indirect buffer), or `nil`.

15 MULE Character Sets and Encodings

Recall that there are two primary ways that text is represented in XEmacs. The *buffer* representation sees the text as a series of bytes (Bufbytes), with a variable number of bytes used per character. The *character* representation sees the text as a series of integers (Emchars), one per character. The character representation is a cleaner representation from a theoretical standpoint, and is thus used in many cases when lots of manipulations on a string need to be done. However, the buffer representation is the standard representation used in both Lisp strings and buffers, and because of this, it is the "default" representation that text comes in. The reason for using this representation is that it's compact and is compatible with ASCII.

15.1 Character Sets

A character set (or *charset*) is an ordered set of characters. A particular character in a charset is indexed using one or more *position codes*, which are non-negative integers. The number of position codes needed to identify a particular character in a charset is called the *dimension* of the charset. In XEmacs/Mule, all charsets have dimension 1 or 2, and the size of all charsets (except for a few special cases) is either 94, 96, 94 by 94, or 96 by 96. The range of position codes used to index characters from any of these types of character sets is as follows:

```
Charset type           Position code 1        Position code 2
------------------------------------------------------------
94                     33 - 126               N/A
96                     32 - 127               N/A
94x94                  33 - 126               33 - 126
96x96                  32 - 127               32 - 127
```

Note that in the above cases position codes do not start at an expected value such as 0 or 1. The reason for this will become clear later.

For example, Latin-1 is a 96-character charset, and JISX0208 (the Japanese national character set) is a 94x94-character charset.

[Note that, although the ranges above define the *valid* position codes for a charset, some of the slots in a particular charset may in fact be empty. This is the case for JISX0208, for example, where (e.g.) all the slots whose first position code is in the range 118 - 127 are empty.]

There are three charsets that do not follow the above rules. All of them have one dimension, and have ranges of position codes as follows:

```
Charset name           Position code 1
------------------------------------
ASCII                  0 - 127
Control-1              0 - 31
Composite              0 - some large number
```

(The upper bound of the position code for composite characters has not yet been determined, but it will probably be at least 16,383).

ASCII is the union of two subsidiary character sets: Printing-ASCII (the printing ASCII character set, consisting of position codes 33 - 126, like for a standard 94-character charset) and Control-ASCII (the non-printing characters that would appear in a binary file with codes 0 - 32 and 127).

Control-1 contains the non-printing characters that would appear in a binary file with codes 128 - 159.

Composite contains characters that are generated by overstriking one or more characters from other charsets.

Note that some characters in ASCII, and all characters in Control-1, are *control* (non-printing) characters. These have no printed representation but instead control some other function of the printing (e.g. TAB or 8 moves the current character position to the next tab stop). All other characters in all charsets are *graphic* (printing) characters.

When a binary file is read in, the bytes in the file are assigned to character sets as follows:

```
Bytes           Character set           Range
--------------------------------------------------
0 - 127         ASCII                   0 - 127
128 - 159       Control-1               0 - 31
160 - 255       Latin-1                 32 - 127
```

This is a bit ad-hoc but gets the job done.

15.2 Encodings

An *encoding* is a way of numerically representing characters from one or more character sets. If an encoding only encompasses one character set, then the position codes for the characters in that character set could be used directly. This is not possible, however, if more than one character set is to be used in the encoding.

For example, the conversion detailed above between bytes in a binary file and characters is effectively an encoding that encompasses the three character sets ASCII, Control-1, and Latin-1 in a stream of 8-bit bytes.

Thus, an encoding can be viewed as a way of encoding characters from a specified group of character sets using a stream of bytes, each of which contains a fixed number of bits (but not necessarily 8, as in the common usage of "byte").

Here are descriptions of a couple of common encodings:

15.2.1 Japanese EUC (Extended Unix Code)

This encompasses the character sets Printing-ASCII, Japanese-JISSX0201, and Japanese-JISX0208-Kana (half-width katakana, the right half of JISX0201). It uses 8-bit bytes.

Note that Printing-ASCII and Japanese-JISX0201-Kana are 94-character charsets, while Japanese-JISX0208 is a 94x94-character charset.

The encoding is as follows:

```
Character set            Representation (PC=position-code)
-------------            --------------
Printing-ASCII           PC1
Japanese-JISX0201-Kana   0x8E        | PC1 + 0x80
Japanese-JISX0208        PC1 + 0x80 | PC2 + 0x80
Japanese-JISX0212        PC1 + 0x80 | PC2 + 0x80
```

15.2.2 JIS7

This encompasses the character sets Printing-ASCII, Japanese-JISX0201-Roman (the left half of JISX0201; this character set is very similar to Printing-ASCII and is a 94-character charset), Japanese-JISX0208, and Japanese-JISX0201-Kana. It uses 7-bit bytes.

Unlike Japanese EUC, this is a *modal* encoding, which means that there are multiple states that the encoding can be in, which affect how the bytes are to be interpreted. Special sequences of bytes (called *escape sequences*) are used to change states.

The encoding is as follows:

```
Character set              Representation (PC=position-code)
-------------              --------------
Printing-ASCII             PC1
Japanese-JISX0201-Roman    PC1
Japanese-JISX0201-Kana     PC1
Japanese-JISX0208          PC1 PC2


Escape sequence   ASCII equivalent   Meaning
---------------   ----------------   -------
0x1B 0x28 0x4A    ESC ( J            invoke Japanese-JISX0201-Roman
0x1B 0x28 0x49    ESC ( I            invoke Japanese-JISX0201-Kana
0x1B 0x24 0x42    ESC $ B            invoke Japanese-JISX0208
0x1B 0x28 0x42    ESC ( B            invoke Printing-ASCII
```

Initially, Printing-ASCII is invoked.

15.3 Internal Mule Encodings

In XEmacs/Mule, each character set is assigned a unique number, called a *leading byte*. This is used in the encodings of a character. Leading bytes are in the range 0x80 - 0xFF (except for ASCII, which has a leading byte of 0), although some leading bytes are reserved.

Charsets whose leading byte is in the range 0x80 - 0x9F are called *official* and are used for built-in charsets. Other charsets are called *private* and have leading bytes in the range 0xA0 - 0xFF; these are user-defined charsets.

More specifically:

```
Character set           Leading byte
-------------           ------------
ASCII                   0
Composite               0x80
Dimension-1 Official    0x81 - 0x8D
                          (0x8E is free)
Control-1               0x8F
Dimension-2 Official    0x90 - 0x99
                          (0x9A - 0x9D are free;
                           0x9E and 0x9F are reserved)
Dimension-1 Private     0xA0 - 0xEF
Dimension-2 Private     0xF0 - 0xFF
```

There are two internal encodings for characters in XEmacs/Mule. One is called *string encoding* and is an 8-bit encoding that is used for representing characters in a buffer or string. It uses 1 to 4 bytes per character. The other is called *character encoding* and is a 19-bit encoding that is used for representing characters individually in a variable.

(In the following descriptions, we'll ignore composite characters for the moment. We also give a general (structural) overview first, followed later by the exact details.)

15.3.1 Internal String Encoding

ASCII characters are encoded using their position code directly. Other characters are encoded using their leading byte followed by their position code(s) with the high bit set. Characters in private character sets have their leading byte prefixed with a *leading byte prefix*, which is either 0x9E or 0x9F. (No character sets are ever assigned these leading bytes.) Specifically:

```
Character set           Encoding (PC=position-code, LB=leading-byte)
-------------           --------
ASCII                   PC-1 |
Control-1               LB   |  PC1 + 0xA0 |
Dimension-1 official    LB   |  PC1 + 0x80 |
Dimension-1 private     0x9E |  LB         | PC1 + 0x80 |
Dimension-2 official    LB   |  PC1 + 0x80 | PC2 + 0x80 |
Dimension-2 private     0x9F |  LB         | PC1 + 0x80 | PC2 + 0x80
```

The basic characteristic of this encoding is that the first byte of all characters is in the range 0x00 - 0x9F, and the second and following bytes of all characters is in the range 0xA0 - 0xFF. This means that it is impossible to get out of sync, or more specifically:

1. Given any byte position, the beginning of the character it is within can be determined in constant time.
2. Given any byte position at the beginning of a character, the beginning of the next character can be determined in constant time.
3. Given any byte position at the beginning of a character, the beginning of the previous character can be determined in constant time.
4. Textual searches can simply treat encoded strings as if they were encoded in a one-byte-per-character fashion rather than the actual multi-byte encoding.

None of the standard non-modal encodings meet all of these conditions. For example, EUC satisfies only (2) and (3), while Shift-JIS and Big5 (not yet described) satisfy only (2). (All non-modal encodings must satisfy (2), in order to be unambiguous.)

15.3.2 Internal Character Encoding

One 19-bit word represents a single character. The word is separated into three fields:

```
Bit number:     18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
                <------------> <------------------> <------------------>
Field:                1                  2                    3
```

Note that fields 2 and 3 hold 7 bits each, while field 1 holds 5 bits.

```
Character set          Field 1        Field 2        Field 3
-------------          -------        -------        -------
ASCII                    0              0              PC1
   range:                                                (00 - 7F)
Control-1                0              1              PC1
   range:                                                (00 - 1F)
Dimension-1 official     0            LB - 0x80        PC1
   range:                                  (01 - 0D)     (20 - 7F)
Dimension-1 private      0            LB - 0x80        PC1
   range:                                  (20 - 6F)     (20 - 7F)
Dimension-2 official   LB - 0x8F        PC1            PC2
   range:                   (01 - 0A)      (20 - 7F)     (20 - 7F)
Dimension-2 private    LB - 0xE1        PC1            PC2
   range:                   (0F - 1E)      (20 - 7F)     (20 - 7F)
Composite                0x1F           ?              ?
```

Note that character codes 0 - 255 are the same as the “binary encoding” described above.

15.4 CCL

```
CCL PROGRAM SYNTAX:
        CCL_PROGRAM := (CCL_MAIN_BLOCK
                        [ CCL_EOF_BLOCK ])

        CCL_MAIN_BLOCK := CCL_BLOCK
        CCL_EOF_BLOCK := CCL_BLOCK

        CCL_BLOCK := STATEMENT | (STATEMENT [STATEMENT ...])
        STATEMENT :=
                SET | IF | BRANCH | LOOP | REPEAT | BREAK
                | READ | WRITE

        SET := (REG = EXPRESSION) | (REG SELF_OP EXPRESSION)
               | INT-OR-CHAR

        EXPRESSION := ARG | (EXPRESSION OP ARG)

        IF := (if EXPRESSION CCL_BLOCK CCL_BLOCK)
        BRANCH := (branch EXPRESSION CCL_BLOCK [CCL_BLOCK ...])
        LOOP := (loop STATEMENT [STATEMENT ...])
        BREAK := (break)
        REPEAT := (repeat)
                | (write-repeat [REG | INT-OR-CHAR | string])
                | (write-read-repeat REG [INT-OR-CHAR | string | ARRAY]?)
        READ := (read REG) | (read REG REG)
                | (read-if REG ARITH_OP ARG CCL_BLOCK CCL_BLOCK)
                | (read-branch REG CCL_BLOCK [CCL_BLOCK ...])
        WRITE := (write REG) | (write REG REG)
                | (write INT-OR-CHAR) | (write STRING) | STRING
                | (write REG ARRAY)
```

```
        END := (end)

        REG := r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7
        ARG := REG | INT-OR-CHAR
        OP :=   + | - | * | / | % | & | '|' | ^ | << | >> | <8 | >8 | //
                | < | > | == | <= | >= | !=
        SELF_OP :=
                += | -= | *= | /= | %= | &= | '|=' | ^= | <<= | >>=
        ARRAY := '[' INT-OR-CHAR ... ']'
        INT-OR-CHAR := INT | CHAR

MACHINE CODE:

The machine code consists of a vector of 32-bit words.
The first such word specifies the start of the EOF section of the code;
this is the code executed to handle any stuff that needs to be done
(e.g. designating back to ASCII and left-to-right mode) after all
other encoded/decoded data has been written out.  This is not used for
charset CCL programs.

REGISTER: 0..7  -- refered by RRR or rrr

OPERATOR BIT FIELD (27-bit): XXXXXXXXXXXXXXX RRR TTTTT
        TTTTT (5-bit): operator type
        RRR (3-bit): register number
        XXXXXXXXXXXXXXXX (15-bit):
                CCCCCCCCCCCCCCC: constant or address
                000000000000rrr: register number

AAAA:   00000 +
        00001 -
        00010 *
        00011 /
        00100 %
        00101 &
        00110 |
        00111 ~

        01000 <<
        01001 >>
        01010 <8
        01011 >8
        01100 //
        01101 not used
        01110 not used
        01111 not used

        10000 <
        10001 >
```

```
        10010 ==
        10011 <=
        10100 >=
        10101 !=

OPERATORS:      TTTTT RRR XX..

SetCS:          00000 RRR C...C         RRR = C...C
SetCL:          00001 RRR .....         RRR = c...c
                c.............c
SetR:           00010 RRR ..rrr         RRR = rrr
SetA:           00011 RRR ..rrr         RRR = array[rrr]
                C.............C         size of array = C...C
                c.............c         contents = c...c

Jump:           00100 000 c...c         jump to c...c
JumpCond:       00101 RRR c...c         if (!RRR) jump to c...c
WriteJump:      00110 RRR c...c         Write1 RRR, jump to c...c
WriteReadJump:  00111 RRR c...c         Write1, Read1 RRR, jump to c...c
WriteCJump:     01000 000 c...c         Write1 C...C, jump to c...c
                C...C
WriteCReadJump: 01001 RRR c...c         Write1 C...C, Read1 RRR,
                C.............C         and jump to c...c
WriteSJump:     01010 000 c...c         WriteS, jump to c...c
                C.............C
                S.............S
                ...
WriteSReadJump: 01011 RRR c...c         WriteS, Read1 RRR, jump to c...c
                C.............C
                S.............S
                ...
WriteAReadJump: 01100 RRR c...c         WriteA, Read1 RRR, jump to c...c
                C.............C         size of array = C...C
                c.............c         contents = c...c
                ...
Branch:         01101 RRR C...C         if (RRR >= 0 && RRR < C..)
                c.............c         branch to (RRR+1)th address
Read1:          01110 RRR ...           read 1-byte to RRR
Read2:          01111 RRR ..rrr         read 2-byte to RRR and rrr
ReadBranch:     10000 RRR C...C         Read1 and Branch
                c.............c
                ...
Write1:         10001 RRR .....         write 1-byte RRR
Write2:         10010 RRR ..rrr         write 2-byte RRR and rrr
WriteC:         10011 000 .....         write 1-char C...CC
                C.............C
WriteS:         10100 000 .....         write C..-byte of string
                C.............C
                S.............S
```

```
                ...
WriteA:         10101 RRR .....         write array[RRR]
                C.............C         size of array = C...C
                c.............c         contents = c...c
                ...
End:            10110 000 .....         terminate the execution

SetSelfCS:      10111 RRR C...C         RRR AAAAA= C...C
                ..........AAAAA
SetSelfCL:      11000 RRR .....         RRR AAAAA= c...c
                c.............c
                ..........AAAAA
SetSelfR:       11001 RRR ..Rrr         RRR AAAAA= rrr
                ..........AAAAA
SetExprCL:      11010 RRR ..Rrr         RRR = rrr AAAAA c...c
                c.............c
                ..........AAAAA
SetExprR:       11011 RRR ..rrr         RRR = rrr AAAAA Rrr
                ............Rrr
                ..........AAAAA
JumpCondC:      11100 RRR c...c         if !(RRR AAAAA C..) jump to c...c
                C.............C
                ..........AAAAA
JumpCondR:      11101 RRR c...c         if !(RRR AAAAA rrr) jump to c...c
                ............rrr
                ..........AAAAA
ReadJumpCondC:  11110 RRR c...c         Read1 and JumpCondC
                C.............C
                ..........AAAAA
ReadJumpCondR:  11111 RRR c...c         Read1 and JumpCondR
                ............rrr
                ..........AAAAA
```

16 The Lisp Reader and Compiler

Not yet documented.

17 Lstreams

An *lstream* is an internal Lisp object that provides a generic buffering stream implementation. Conceptually, you send data to the stream or read data from the stream, not caring what's on the other end of the stream. The other end could be another stream, a file descriptor, a stdio stream, a fixed block of memory, a reallocating block of memory, etc. The main purpose of the stream is to provide a standard interface and to do buffering. Macros are defined to read or write characters, so the calling functions do not have to worry about blocking data together in order to achieve efficiency.

17.1 Creating an Lstream

Lstreams come in different types, depending on what is being interfaced to. Although the primitive for creating new lstreams is `Lstream_new()`, generally you do not call this directly. Instead, you call some type-specific creation function, which creates the lstream and initializes it as appropriate for the particular type.

All lstream creation functions take a *mode* argument, specifying what mode the lstream should be opened as. This controls whether the lstream is for input and output, and optionally whether data should be blocked up in units of MULE characters. Note that some types of lstreams can only be opened for input; others only for output; and others can be opened either way. #### Richard Mlynarik thinks that there should be a strict separation between input and output streams, and he's probably right.

mode is a string, one of

`"r"` Open for reading.

`"w"` Open for writing.

`"rc"` Open for reading, but "read" never returns partial MULE characters.

`"wc"` Open for writing, but never writes partial MULE characters.

17.2 Lstream Types

stdio

filedesc

lisp-string

fixed-buffer

resizing-buffer

dynarr

lisp-buffer

print

decoding

encoding

17.3 Lstream Functions

`Lstream *` **Lstream_new** (`Lstream_implementation` **imp*, `CONST char` **mode*) — Function

Allocate and return a new Lstream. This function is not really meant to be called directly; rather, each stream type should provide its own stream creation function, which creates the stream and does any other necessary creation stuff (e.g. opening a file).

`void` **Lstream_set_buffering** (`Lstream` **lstr*, `Lstream_buffering` *buffering*, `int` *buffering_size*) — Function

Change the buffering of a stream. See 'lstream.h'. By default the buffering is `STREAM_BLOCK_BUFFERED`.

`int` **Lstream_flush** (`Lstream` **lstr*) — Function

Flush out any pending unwritten data in the stream. Clear any buffered input data. Returns 0 on success, -1 on error.

`int` **Lstream_putc** (`Lstream` **stream*, `int` *c*) — Macro

Write out one byte to the stream. This is a macro and so it is very efficient. The *c* argument is only evaluated once but the *stream* argument is evaluated more than once. Returns 0 on success, -1 on error.

`int` **Lstream_getc** (`Lstream` **stream*) — Macro

Read one byte from the stream. This is a macro and so it is very efficient. The *stream* argument is evaluated more than once. Return value is -1 for EOF or error.

`void` **Lstream_ungetc** (`Lstream` **stream*, `int` *c*) — Macro

Push one byte back onto the input queue. This will be the next byte read from the stream. Any number of bytes can be pushed back and will be read in the reverse order they were pushed back – most recent first. (This is necessary for consistency – if there are a number of bytes that have been unread and I read and unread a byte, it needs to be the first to be read again.) This is a macro and so it is very efficient. The *c* argument is only evaluated once but the *stream* argument is evaluated more than once.

`int` **Lstream_fputc** (`Lstream` **stream*, `int` *c*) — Function
`int` **Lstream_fgetc** (`Lstream` **stream*) — Function
`void` **Lstream_fungetc** (`Lstream` **stream*, `int` *c*) — Function

Function equivalents of the above macros.

`int` **Lstream_read** (`Lstream` **stream*, `void` **data*, `int` *size*) — Function

Read *size* bytes of *data* from the stream. Return the number of bytes read. 0 means EOF. -1 means an error occurred and no bytes were read.

int **Lstream_write** (Lstream **stream*, void **data*, int *size*) Function

Write *size* bytes of *data* to the stream. Return the number of bytes written. -1 means an error occurred and no bytes were written.

void **Lstream_unread** (Lstream **stream*, void **data*, int *size*) Function

Push back *size* bytes of *data* onto the input queue. The next call to `Lstream_read()` with the same size will read the same bytes back. Note that this will be the case even if there is other pending unread data.

int **Lstream_close** (Lstream **stream*) Function

Close the stream. All data will be flushed out.

void **Lstream_reopen** (Lstream **stream*) Function

Reopen a closed stream. This enables I/O on it again. This is not meant to be called except from a wrapper routine that reinitializes variables and such – the close routine may well have freed some necessary storage structures, for example.

void **Lstream_rewind** (Lstream **stream*) Function

Rewind the stream to the beginning.

17.4 Lstream Methods

int **reader** (Lstream **stream*, unsigned char **data*, int *size*) Lstream Method

Read some data from the stream's end and store it into *data*, which can hold *size* bytes. Return the number of bytes read. A return value of 0 means no bytes can be read at this time. This may be because of an EOF, or because there is a granularity greater than one byte that the stream imposes on the returned data, and *size* is less than this granularity. (This will happen frequently for streams that need to return whole characters, because `Lstream_read()` calls the reader function repeatedly until it has the number of bytes it wants or until 0 is returned.) The lstream functions do not treat a 0 return as EOF or do anything special; however, the calling function will interpret any 0 it gets back as EOF. This will normally not happen unless the caller calls `Lstream_read()` with a very small size.

This function can be `NULL` if the stream is output-only.

int **writer** (Lstream **stream*, CONST unsigned char **data*, int *size*) Lstream Method

Send some data to the stream's end. Data to be sent is in *data* and is *size* bytes. Return the number of bytes sent. This function can send and return fewer bytes than is passed in; in that case, the function will just be called again until there is no data left or 0 is returned. A return value of 0 means that no more data can be currently stored, but there is no error; the data will be squirreled away until the writer can accept data. (This is useful, e.g., if you're dealing with a non-blocking file descriptor and are getting `EWOULDBLOCK` errors.) This function can be `NULL` if the stream is input-only.

`int` **rewinder** (`Lstream *`*stream*) — Lstream Method

Rewind the stream. If this is `NULL`, the stream is not seekable.

`int` **seekable_p** (`Lstream *`*stream*) — Lstream Method

Indicate whether this stream is seekable – i.e. it can be rewound. This method is ignored if the stream does not have a rewind method. If this method is not present, the result is determined by whether a rewind method is present.

`int` **flusher** (`Lstream *`*stream*) — Lstream Method

Perform any additional operations necessary to flush the data in this stream.

`int` **pseudo_closer** (`Lstream *`*stream*) — Lstream Method

`int` **closer** (`Lstream *`*stream*) — Lstream Method

Perform any additional operations necessary to close this stream down. May be `NULL`. This function is called when `Lstream_close()` is called or when the stream is garbage-collected. When this function is called, all pending data in the stream will already have been written out.

`Lisp_Object` **marker** (`Lisp_Object` *lstream*, `void (*`*markfun*`) (Lisp_Object))` — Lstream Method

Mark this object for garbage collection. Same semantics as a standard `Lisp_Object` marker. This function can be `NULL`.

18 Consoles; Devices; Frames; Windows

18.1 Introduction to Consoles; Devices; Frames; Windows

A window-system window that you see on the screen is called a *frame* in Emacs terminology. Each frame is subdivided into one or more non-overlapping panes, called (confusingly) *windows*. Each window displays the text of a buffer in it. (See above on Buffers.) Note that buffers and windows are independent entities: Two or more windows can be displaying the same buffer (potentially in different locations), and a buffer can be displayed in no windows.

A single display screen that contains one or more frames is called a *display*. Under most circumstances, there is only one display. However, more than one display can exist, for example if you have a *multi-headed* console, i.e. one with a single keyboard but multiple displays. (Typically in such a situation, the various displays act like one large display, in that the mouse is only in one of them at a time, and moving the mouse off of one moves it into another.) In some cases, the different displays will have different characteristics, e.g. one color and one mono.

XEmacs can display frames on multiple displays. It can even deal simultaneously with frames on multiple keyboards (called *consoles* in XEmacs terminology). Here is one case where this might be useful: You are using XEmacs on your workstation at work, and leave it running. Then you go home and dial in on a TTY line, and you can use the already-running XEmacs process to display another frame on your local TTY.

Thus, there is a hierarchy console `->` display `->` frame `->` window. There is a separate Lisp object type for each of these four concepts. Furthermore, there is logically a *selected console*, *selected display*, *selected frame*, and *selected window*. Each of these objects is distinguished in various ways, such as being the default object for various functions that act on objects of that type. Note that every containing object rememembers the “selected” object among the objects that it contains: e.g. not only is there a selected window, but every frame remembers the last window in it that was selected, and changing the selected frame causes the remembered window within it to become the selected window. Similar relationships apply for consoles to devices and devices to frames.

18.2 Point

Recall that every buffer has a current insertion position, called *point*. Now, two or more windows may be displaying the same buffer, and the text cursor in the two windows (i.e. `point`) can be in two different places. You may ask, how can that be, since each buffer has only one value of `point`? The answer is that each window also has a value of `point` that is squirreled away in it. There is only one selected window, and the value of “point” in that buffer corresponds to that window. When the selected window is changed from one window to another displaying the same buffer, the old value of `point` is stored into the old window’s “point” and the value of `point` from the new window is retrieved and made the value of `point` in the buffer. This means that `window-point` for the selected window is potentially inaccurate, and if you want to retrieve the correct value of `point` for a window, you must special-case on the selected window and retrieve the buffer’s point instead. This is related to why `save-window-excursion` does not save the selected window’s value of `point`.

18.3 Window Hierarchy

If a frame contains multiple windows (panes), they are always created by splitting an existing window along the horizontal or vertical axis. Terminology is a bit confusing here: to *split a window horizontally* means to create two side-by-side windows, i.e. to make a *vertical* cut in a window. Likewise, to *split a window vertically* means to create two windows, one above the other, by making a *horizontal* cut.

If you split a window and then split again along the same axis, you will end up with a number of panes all arranged along the same axis. The precise way in which the splits were made should not be important, and this is reflected internally. Internally, all windows are arranged in a tree, consisting of two types of windows, *combination* windows (which have children, and are covered completely by those children) and *leaf* windows, which have no children and are visible. Every combination window has two or more children, all arranged along the same axis. There are (logically) two subtypes of windows, depending on whether their children are horizontally or vertically arrayed. There is always one root window, which is either a leaf window (if the frame contains only one window) or a combination window (if the frame contains more than one window). In the latter case, the root window will have two or more children, either horizontally or vertically arrayed, and each of those children will be either a leaf window or another combination window.

Here are some rules:

1. Horizontal combination windows can never have children that are horizontal combination windows; same for vertical.
2. Only leaf windows can be split (obviously) and this splitting does one of two things: (a) turns the leaf window into a combination window and creates two new leaf children, or (b) turns the leaf window into one of the two new leaves and creates the other leaf. Rule (1) dictates which of these two outcomes happens.
3. Every combination window must have at least two children.
4. Leaf windows can never become combination windows. They can be deleted, however. If this results in a violation of (3), the parent combination window also gets deleted.
5. All functions that accept windows must be prepared to accept combination windows, and do something sane (e.g. signal an error if so). Combination windows *do* escape to the Lisp level.
6. All windows have three fields governing their contents: these are *hchild* (a list of horizontally-arrayed children), *vchild* (a list of vertically-arrayed children), and *buffer* (the buffer contained in a leaf window). Exactly one of these will be non-nil. Remember that *horizontally-arrayed* means "side-by-side" and *vertically-arrayed* means *one above the other*.
7. Leaf windows also have markers in their `start` (the first buffer position displayed in the window) and `pointm` (the window's stashed value of `point` – see above) fields, while combination windows have nil in these fields.
8. The list of children for a window is threaded through the `next` and `prev` fields of each child window.
9. **Deleted windows can be undeleted**. This happens as a result of restoring a window configuration, and is unlike frames, displays, and consoles, which, once deleted, can

never be restored. Deleting a window does nothing except set a special `dead` bit to 1 and clear out the `next`, `prev`, `hchild`, and `vchild` fields, for GC purposes.

10. Most frames actually have two top-level windows – one for the minibuffer and one (the *root*) for everything else. The modeline (if present) separates these two. The `next` field of the root points to the minibuffer, and the `prev` field of the minibuffer points to the root. The other `next` and `prev` fields are `nil`, and the frame points to both of these windows. Minibuffer-less frames have no minibuffer window, and the `next` and `prev` of the root window are `nil`. Minibuffer-only frames have no root window, and the `next` of the minibuffer window is `nil` but the `prev` points to itself. (#### This is an artifact that should be fixed.)

18.4 The Window Object

Windows have the following accessible fields:

`frame` The frame that this window is on.

`mini_p` Non-`nil` if this window is a minibuffer window.

`buffer` The buffer that the window is displaying. This may change often during the life of the window.

`dedicated`
: Non-`nil` if this window is dedicated to its buffer.

`pointm` This is the value of point in the current buffer when this window is selected; when it is not selected, it retains its previous value.

`start` The position in the buffer that is the first character to be displayed in the window.

`force_start`
: If this flag is non-`nil`, it says that the window has been scrolled explicitly by the Lisp program. This affects what the next redisplay does if point is off the screen: instead of scrolling the window to show the text around point, it moves point to a location that is on the screen.

`last_modified`
: The `modified` field of the window's buffer, as of the last time a redisplay completed in this window.

`last_point`
: The buffer's value of point, as of the last time a redisplay completed in this window.

`left` This is the left-hand edge of the window, measured in columns. (The leftmost column on the screen is column 0.)

`top` This is the top edge of the window, measured in lines. (The top line on the screen is line 0.)

`height` The height of the window, measured in lines.

`width` The width of the window, measured in columns.

`next` This is the window that is the next in the chain of siblings. It is `nil` in a window that is the rightmost or bottommost of a group of siblings.

`prev` This is the window that is the previous in the chain of siblings. It is `nil` in a window that is the leftmost or topmost of a group of siblings.

`parent` Internally, XEmacs arranges windows in a tree; each group of siblings has a parent window whose area includes all the siblings. This field points to a window's parent.

Parent windows do not display buffers, and play little role in display except to shape their child windows. Emacs Lisp programs usually have no access to the parent windows; they operate on the windows at the leaves of the tree, which actually display buffers.

`hscroll` This is the number of columns that the display in the window is scrolled horizontally to the left. Normally, this is 0.

`use_time` This is the last time that the window was selected. The function `get-lru-window` uses this field.

`display_table`
The window's display table, or `nil` if none is specified for it.

`update_mode_line`
Non-`nil` means this window's mode line needs to be updated.

`base_line_number`
The line number of a certain position in the buffer, or `nil`. This is used for displaying the line number of point in the mode line.

`base_line_pos`
The position in the buffer for which the line number is known, or `nil` meaning none is known.

`region_showing`
If the region (or part of it) is highlighted in this window, this field holds the mark position that made one end of that region. Otherwise, this field is `nil`.

19 The Redisplay Mechanism

The redisplay mechanism is one of the most complicated sections of XEmacs, especially from a conceptual standpoint. This is doubly so because, unlike for the basic aspects of the Lisp interpreter, the computer science theories of how to efficiently handle redisplay are not well-developed.

When working with the redisplay mechanism, remember the Golden Rules of Redisplay:

1. It Is Better To Be Correct Than Fast.
2. Thou Shalt Not Run Elisp From Within Redisplay.
3. It Is Better To Be Fast Than Not To Be.

19.1 Critical Redisplay Sections

Within this section, we are defenseless and assume that the following cannot happen:

1. garbage collection
2. Lisp code evaluation
3. frame size changes

We ensure (3) by calling `hold_frame_size_changes()`, which will cause any pending frame size changes to get put on hold till after the end of the critical section. (1) follows automatically if (2) is met. #### Unfortunately, there are some places where Lisp code can be called within this section. We need to remove them.

If `Fsignal()` is called during this critical section, we will `abort()`.

If garbage collection is called during this critical section, we simply return. #### We should abort instead.

If a frame-size change does occur we should probably actually be preempting redisplay.

19.2 Line Start Cache

The traditional scrolling code in Emacs breaks in a variable height world. It depends on the key assumption that the number of lines that can be displayed at any given time is fixed. This led to a complete separation of the scrolling code from the redisplay code. In order to fully support variable height lines, the scrolling code must actually be tightly integrated with redisplay. Only redisplay can determine how many lines will be displayed on a screen for any given starting point.

What is ideally wanted is a complete list of the starting buffer position for every possible display line of a buffer along with the height of that display line. Maintaining such a full list would be very expensive. We settle for having it include information for all areas which we happen to generate anyhow (i.e. the region currently being displayed) and for those areas we need to work with.

In order to ensure that the cache accurately represents what redisplay would actually show, it is necessary to invalidate it in many situations. If the buffer changes, the starting positions may no longer be correct. If a face or an extent has changed then the line heights may have altered. These events happen frequently enough that the cache can end up being constantly disabled. With this potentially constant invalidation when is the cache ever useful?

Even if the cache is invalidated before every single usage, it is necessary. Scrolling often requires knowledge about display lines which are actually above or below the visible region. The cache provides a convenient light-weight method of storing this information for multiple display regions. This knowledge is necessary for the scrolling code to always obey the First Golden Rule of Redisplay.

If the cache already contains all of the information that the scrolling routines happen to need so that it doesn't have to go generate it, then we are able to obey the Third Golden Rule of Redisplay. The first thing we do to help out the cache is to always add the displayed region. This region had to be generated anyway, so the cache ends up getting the information basically for free. In those cases where a user is simply scrolling around viewing a buffer there is a high probability that this is sufficient to always provide the needed information. The second thing we can do is be smart about invalidating the cache.

TODO – Be smart about invalidating the cache. Potential places:

- Insertions at end-of-line which don't cause line-wraps do not alter the starting positions of any display lines. These types of buffer modifications should not invalidate the cache. This is actually a large optimization for redisplay speed as well.
- Buffer modifications frequently only affect the display of lines at and below where they occur. In these situations we should only invalidate the part of the cache starting at where the modification occurs.

In case you're wondering, the Second Golden Rule of Redisplay is not applicable.

20 Extents

20.1 Introduction to Extents

Extents are regions over a buffer, with a start and an end position denoting the region of the buffer included in the extent. In addition, either end can be closed or open, meaning that the endpoint is or is not logically included in the extent. Insertion of a character at a closed endpoint causes the character to go inside the extent; insertion at an open endpoint causes the character to go outside.

Extent endpoints are stored using memory indices (see '`insdel.c`'), to minimize the amount of adjusting that needs to be done when characters are inserted or deleted.

(Formerly, extent endpoints at the gap could be either before or after the gap, depending on the open/closedness of the endpoint. The intent of this was to make it so that insertions would automatically go inside or out of extents as necessary with no further work needing to be done. It didn't work out that way, however, and just ended up complexifying and buggifying all the rest of the code.)

20.2 Extent Ordering

Extents are compared using memory indices. There are two orderings for extents and both orders are kept current at all times. The normal or *display* order is as follows:

```
Extent A is ``less than'' extent B, that is, earlier in the display order,
if:    A-start < B-start,
or if: A-start = B-start, and A-end > B-end
```

So if two extents begin at the same position, the larger of them is the earlier one in the display order (`EXTENT_LESS` is true).

For the e-order, the same thing holds:

```
Extent A is ``less than'' extent B in e-order, that is, later in the buffer,
if:    A-end < B-end,
or if: A-end = B-end, and A-start > B-start
```

So if two extents end at the same position, the smaller of them is the earlier one in the e-order (`EXTENT_E_LESS` is true).

The display order and the e-order are complementary orders: any theorem about the display order also applies to the e-order if you swap all occurrences of "display order" and "e-order", "less than" and "greater than", and "extent start" and "extent end".

20.3 Format of the Extent Info

An extent-info structure consists of a list of the buffer or string's extents and a *stack of extents* that lists all of the extents over a particular position. The stack-of-extents info is used for optimization purposes – it basically caches some info that might be expensive to compute. Certain otherwise hard computations are easy given the stack of extents over a particular position, and if the stack of extents over a nearby position is known (because it was calculated at some prior point in time), it's easy to move the stack of extents to the proper position.

Given that the stack of extents is an optimization, and given that it requires memory, a string's stack of extents is wiped out each time a garbage collection occurs. Therefore, any time you retrieve the stack of extents, it might not be there. If you need it to be there, use the `_force` version.

Similarly, a string may or may not have an extent_info structure. (Generally it won't if there haven't been any extents added to the string.) So use the `_force` version if you need the extent_info structure to be there.

A list of extents is maintained as a double gap array: one gap array is ordered by start index (the *display order*) and the other is ordered by end index (the *e-order*). Note that positions in an extent list should logically be conceived of as referring *to* a particular extent (as is the norm in programs) rather than sitting between two extents. Note also that callers of these functions should not be aware of the fact that the extent list is implemented as an array, except for the fact that positions are integers (this should be generalized to handle integers and linked list equally well).

20.4 Zero-Length Extents

Extents can be zero-length, and will end up that way if their endpoints are explicitly set that way or if their detachable property is nil and all the text in the extent is deleted. (The exception is open-open zero-length extents, which are barred from existing because there is no sensible way to define their properties. Deletion of the text in an open-open extent causes it to be converted into a closed-open extent.) Zero-length extents are primarily used to represent annotations, and behave as follows:

1. Insertion at the position of a zero-length extent expands the extent if both endpoints are closed; goes after the extent if it is closed-open; and goes before the extent if it is open-closed.
2. Deletion of a character on a side of a zero-length extent whose corresponding endpoint is closed causes the extent to be detached if it is detachable; if the extent is not detachable or the corresponding endpoint is open, the extent remains in the buffer, moving as necessary.

Note that closed-open, non-detachable zero-length extents behave exactly like markers and that open-closed, non-detachable zero-length extents behave like the "point-type" marker in Mule.

20.5 Mathematics of Extent Ordering

The extents in a buffer are ordered by "display order" because that is that order that the redisplay mechanism needs to process them in. The e-order is an auxiliary ordering used to facilitate operations over extents. The operations that can be performed on the ordered list of extents in a buffer are

1. Locate where an extent would go if inserted into the list.
2. Insert an extent into the list.
3. Remove an extent from the list.
4. Map over all the extents that overlap a range.

(4) requires being able to determine the first and last extents that overlap a range.

NOTE: *overlap* is used as follows:

- two ranges overlap if they have at least one point in common. Whether the endpoints are open or closed makes a difference here.
- a point overlaps a range if the point is contained within the range; this is equivalent to treating a point P as the range $[P, P]$.
- In the case of an *extent* overlapping a point or range, the extent is normally treated as having closed endpoints. This applies consistently in the discussion of stacks of extents and such below. Note that this definition of overlap is not necessarily consistent with the extents that `map-extents` maps over, since `map-extents` sometimes pays attention to whether the endpoints of an extents are open or closed. But for our purposes, it greatly simplifies things to treat all extents as having closed endpoints.

First, define $>$, $<$, $<=$, etc. as applied to extents to mean comparison according to the display order. Comparison between an extent E and an index I means comparison between E and the range $[I, I]$.

Also define $e>$, $e<$, $e<=$, etc. to mean comparison according to the e-order.

For any range R, define $R(0)$ to be the starting index of the range and $R(1)$ to be the ending index of the range.

For any extent E, define $E(next)$ to be the extent directly following E, and $E(prev)$ to be the extent directly preceding E. Assume $E(next)$ and $E(prev)$ can be determined from E in constant time. (This is because we store the extent list as a doubly linked list.)

Similarly, define $E(e-next)$ and $E(e-prev)$ to be the extents directly following and preceding E in the e-order.

Now:

Let R be a range. Let F be the first extent overlapping R. Let L be the last extent overlapping R.

Theorem 1: $R(1)$ lies between L and $L(next)$, i.e. $L<=R(1)<L(next)$.

This follows easily from the definition of display order. The basic reason that this theorem applies is that the display order sorts by increasing starting index.

Therefore, we can determine L just by looking at where we would insert $R(1)$ into the list, and if we know F and are moving forward over extents, we can easily determine when we've hit L by comparing the extent we're at to $R(1)$.

    Theorem 2: $F(e-prev)\ e<\ [1,\ R(0)]\ e<=\ F$.

This is the analog of Theorem 1, and applies because the e-order sorts by increasing ending index.

Therefore, F can be found in the same amount of time as operation (1), i.e. the time that it takes to locate where an extent would go if inserted into the e-order list.

If the lists were stored as balanced binary trees, then operation (1) would take logarithmic time, which is usually quite fast. However, currently they're stored as simple doubly-linked lists, and instead we do some caching to try to speed things up.

Define a *stack of extents* (or *SOE*) as the set of extents (ordered in the display order) that overlap an index I, together with the SOE's *previous* extent, which is an extent that precedes I in the e-order. (Hopefully there will not be very many extents between I and the previous extent.)

Now:

Let I be an index, let S be the stack of extents on I, let F be the first extent in S, and let P be S's previous extent.

Theorem 3: The first extent in S is the first extent that overlaps any range $[I, J]$.

Proof: Any extent that overlaps $[I, J]$ but does not include I must have a start index $>I$, and thus be greater than any extent in S.

Therefore, finding the first extent that overlaps a range R is the same as finding the first extent that overlaps $R(0)$.

Theorem 4: Let $I2$ be an index such that $I2>I$, and let $F2$ be the first extent that overlaps $I2$. Then, either $F2$ is in S or $F2$ is greater than any extent in S.

Proof: If $F2$ does not include I then its start index is greater than I and thus it is greater than any extent in S, including F. Otherwise, $F2$ includes I and thus is in S, and thus $F2>=F$.

20.6 Extent Fragments

Imagine that the buffer is divided up into contiguous, non-overlapping *runs* of text such that no extent starts or ends within a run (extents that abut the run don't count).

An extent fragment is a structure that holds data about the run that contains a particular buffer position (if the buffer position is at the junction of two runs, the run after the position is used) – the beginning and end of the run, a list of all of the extents in that run, the *merged face* that results from merging all of the faces corresponding to those extents, the begin and end glyphs at the beginning of the run, etc. This is the information that redisplay needs in order to display this run.

Extent fragments have to be very quick to update to a new buffer position when moving linearly through the buffer. They rely on the stack-of-extents code, which does the heavy-duty algorithmic work of determining which extents overly a particular position.

21 Faces and Glyphs

Not yet documented.

22 Specifiers

Not yet documented.

23 Menus

A menu is set by setting the value of the variable `current-menubar` (which may be buffer-local) and then calling `set-menubar-dirty-flag` to signal a change. This will cause the menu to be redrawn at the next redisplay. The format of the data in `current-menubar` is described in '`menubar.c`'.

Internally the data in current-menubar is parsed into a tree of `widget_value's` (defined in '`lwlib.h`'); this is accomplished by the recursive function `menu_item_descriptor_to_widget_value()`, called by `compute_menubar_data()`. Such a tree is deallocated using `free_widget_value()`.

`update_screen_menubars()` is one of the external entry points. This checks to see, for each screen, if that screen's menubar needs to be updated. This is the case if

1. `set-menubar-dirty-flag` was called since the last redisplay. (This function sets the C variable menubar_has_changed.)
2. The buffer displayed in the screen has changed.
3. The screen has no menubar currently displayed.

`set_screen_menubar()` is called for each such screen. This function calls `compute_menubar_data()` to create the tree of widget_value's, then calls `lw_create_widget()`, `lw_modify_all_widgets()`, and/or `lw_destroy_all_widgets()` to create the X-Toolkit widget associated with the menu.

`update_psheets()`, the other external entry point, actually changes the menus being displayed. It uses the widgets fixed by `update_screen_menubars()` and calls various X functions to ensure that the menus are displayed properly.

The menubar widget is set up so that `pre_activate_callback()` is called when the menu is first selected (i.e. mouse button goes down), and `menubar_selection_callback()` is called when an item is selected. `pre_activate_callback()` calls the function in activate-menubar-hook, which can change the menubar (this is described in '`menubar.c`'). If the menubar is changed, `set_screen_menubars()` is called. `menubar_selection_callback()` enqueues a menu event, putting in it a function to call (either `eval` or `call-interactively`) and its argument, which is the callback function or form given in the menu's description.

24 Subprocesses

The fields of a process are:

`name` A string, the name of the process.

`command` A list containing the command arguments that were used to start this process.

`filter` A function used to accept output from the process instead of a buffer, or `nil`.

`sentinel` A function called whenever the process receives a signal, or `nil`.

`buffer` The associated buffer of the process.

`pid` An integer, the Unix process ID.

`childp` A flag, non-`nil` if this is really a child process. It is `nil` for a network connection.

`mark` A marker indicating the position of the end of the last output from this process inserted into the buffer. This is often but not always the end of the buffer.

`kill_without_query`
: If this is non-`nil`, killing XEmacs while this process is still running does not ask for confirmation about killing the process.

`raw_status_low`
`raw_status_high`
: These two fields record 16 bits each of the process status returned by the `wait` system call.

`status` The process status, as `process-status` should return it.

`tick`
`update_tick`
: If these two fields are not equal, a change in the status of the process needs to be reported, either by running the sentinel or by inserting a message in the process buffer.

`pty_flag` Non-`nil` if communication with the subprocess uses a PTY; `nil` if it uses a pipe.

`infd` The file descriptor for input from the process.

`outfd` The file descriptor for output to the process.

`subtty` The file descriptor for the terminal that the subprocess is using. (On some systems, there is no need to record this, so the value is `-1`.)

`tty_name` The name of the terminal that the subprocess is using, or `nil` if it is using pipes.

25 Interface to X Windows

Not yet documented.

Index

All variables, functions, keys, programs, files, and concepts are in this one index.

All names and concepts are permuted, so they appear several times, one for each permutation of the parts of the name. For example, `function-name` would appear as **function-name** and **name, function-**. Key entries are not permuted, however.

F

G

H

I

J

K

L

M

N

O

P

R

S

Short Contents

Table of Contents