Epoch

GNU Emacs for the X Windowing System

Release 3.2 (beta)

Based on GNU Emacs 18.55

June 1990

Copyright © 1989,1990 by

Alan M. Carroll (carroll@cs.uiuc.edu)

Simon M. Kaplan (kaplan@cs.uiuc.edu)

Daniel M. LaLiberte (liberte@cs.uiuc.edu)

This documents release 3.2 of Epoch which is based on GNU Emacs 18.55. Published by ...

1304 W Springfield
Department of Computer Science
University of Illinois, Urbana-Champaign

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the ...University of Illinois, Urbana-Champaign

1. Introduction

Epoch is a modified version of GNU Emacs with several major enhancements:

- Multiple X-Window support
- Marked and attributed regions of text
- Asynchronous communication with other X clients (usually the window manager).
- Mouse-dragging with highlighting to support mouse-based cut-and-paste.
- Access to raw X-Window objects.

Some minor enhancements include various bug fixes and improved option handling.

As much compatability as possible with previous version of GNU Emacs has been maintained. Existing elisp code should run under Epoch, although it will not automatically use the new features of Epoch. Almost all the X support code in GNU Emacs was removed because of incompatabilities. Almost all of the features except the X menu code were reproduced. We believe that the window manager should provide menus (see Section 2.5 [GWM], page 11).

Epoch requires X-Windows to function - it will not run on non-graphics terminals.

Most of this manual is intended for the "Emacs guru" who wants to play with and extend Epoch's functionality. Information intended for "ordinary Emacs users" is found in See Chapter 2 [General Information], page 5.

1.1 Support and Mailing List

We have established an Epoch mailing list. If there is enough interest we will make it into a news-group. We strongly encourage you to join, so as to share ideas about applications of Epoch, code and (hopefully not many) bug reports.

To post to the mailing list send mail to one of these addresses:

```
epoch@cs.uiuc.edu
uunet!uiucdcs!epoch
epoch%cs.uiuc.edu@uiucvmd.bitnet
```

For administrative messages (joining the mailing list, etc), substitute '`epoch`' with '`epoch-request`' in the above.

1.2 Getting Epoch

Epoch is available by anonymous ftp on cs.uiuc.edu. We are negotiating to have it available through other sites, and through uucp mail request. Send mail to epoch-request to find out the current status with this if you do not have anonymous ftp.

Epoch is available on tape: The cost at the time of creating version 3.2 was $175.00 for cartridge tapes. We can currently make 1600 BPI reel tapes and cartridge tapes for SUN and HP 9000 series machines. The price covers the cost of purchasing, making and shipping the tapes. You should contact the epoch-request mailing list to confirm these rates. Payment should be by cheque made payable to the University of Illinois, and sent to:

```
Epoch Distribution, attn: R. Canaday
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana
Illinois 61801, USA.
```

Update tapes are distributed as the same rate, but we can also email diff files, which are usually small enough to make this practical.

1.3 Acknowledgements

Simon would like to thank the Computer Science Department at the University of Illinois, AT&T and the National Science Foundation for their support of parts of this work, and also Philippa and Ronan for putting up with him.

Dan would like to thank Simon and Alan for their fine work, and Clare and Carey for putting him up.

Chris Love has done considerable work in the preparation of version Epoch 3.2.

Susan Hinrichs, Alan's lovely wife, provided emotional, intellectual, and code support.

We would all like to thank Richard Stallman and the Free Software Foundation for GNU Emacs, and Colas Nahaboo and Bull for the GWM window manager. We would also like to thank all those who have provided invaluable advice and assistance in getting Epoch to this stage of development, in particular Denys Duchier of Yale University.

2. General Information

2.1 Getting Started

We assume you are reasonably familiar with GNU Emacs. You can essentially use Epoch just like GNU Emacs.

Epoch is started just like GNU Emacs. It opens it own X-windows, so it should be run in background. One immediately noticable difference is that Epoch always creates *two* windows by default, one for the minibuffer and one for the initial edit buffer. More on this below.

If you wish to use Epoch without having to do any Elisp programming, while still specifying window properties, the following sections are for you.

2.1.1 X Window Program Defaults

Epoch attempts to read the default values associated with X Window programs, or *X defaults*, set up by the user. These values are loaded into an internal database, called the global default-set, which is initially filled in with hard-wired defaults.

The resulting global default-set is stored, and remains static for the rest of the program execution. It serves as a basis for all screen creation, in order to guarantee a value for every property. (The minibuffer has its own default-set, which is used once to create the minibuffer screen.) See Section 3.1 [Screen Properties], page 14 for more background on how X defaults are used.

The X default specifications are determined in the following order:

1. Built-in X defaults.
2. $HOME/.Xdefaults
3. File named by the environment variable XENVIRONMENT
4. If XENVIRONMENT is not set, $HOME/.Xdefaults-<hostname>
5. Resources database (see '`xrdb`')
6. Command line options

The X defaults recognized by Epoch are listed in the table below. The window resource

name/class defaults are special; they default to the program resource name/class respectively (see Section 2.1.2 [Command Line Options], page 8).

Furthermore, the *program* resource name/class command line options are special. They are not put into the internal database - they affect how the database is queried for X default values. When searching for names and classes, the leftmost qualifier is the resource name/class name. The default for the name is the name of the program (normally "epoch") and the default for the class is "Emacs". If these are set by command line options, the new values become the X defaults.

Epoch also accepts class defaults (as distinct from resource defaults), which are exactly as below except that the first letter of each field is capitalized.

`Option` Default Value, Effect

`.screen.background`
White, Edit screen background color

`.minibuf.background`
White, Minibuffer screen background color

`.screen.border.color`
Black, Edit screen border color

`.minibuf.border.color`
Black, Minibuffer screen border color

`.screen.border.width`
2, Edit screen border width

`.minibuf.border.width`
2, Minibuffer screen border width

`.screen.cursor.background`
White, Edit screen X cursor background color

`.minibuf.cursor.background`
White, Minibuffer screen X cursor background color

`.screen.cursor.color`
Black, Edit screen text cursor color

`.minibuf.cursor.color`
Black, Minibuffer screen text cursor color

`.screen.cursor.foreground`
Black, Edit screen X cursor foreground color

`.minibuf.cursor.foreground`
Black, Minibuffer screen X cursor foreground color

`.screen.cursor.glyph`
: 86, Edit screen X cursor glyph

`.minibuf.cursor.glyph`
: 86, Minibuffer screen X cursor glyph

`.screen.internal.border.width`
: 2, Edit screen internal border width

`.minibuf.internal.border.width`
: 2, Minibuffer screen internal border width

`.screen.class`
: <special>, Resource class of the edit screen

`.minibuf.class`
: <special>, Resource class of the minibuffer screen

`.display` : "", X Window display

`.motion` : Off, Whether motion-hints should come through to elisp

`.screen.font`
: "fixed", Edit screen font

`.minibuf.font`
: "fixed", Minibuffer screen font

`.screen.foreground`
: Black, Edit screen foreground color

`.minibuf.foreground`
: Black, Minibuffer screen foreground color

`.screen.geometry`
: "80x24", Geometry of the initial edit screen

`.minibuf.geometry`
: "80x1", Geometry of the minibuffer screen

`.screen.name`
: "Edit", Name of the edit screen

`.minibuf.name`
: "Minibuffer", Name of the minibuffer screen

`.screen.resource`
: <special>, Resource name of the edit screen

`.minibuf.resource`
: <special>, Resource name of the minibuffer screen

`.screen.reverse`
: False, Reverse colors for edit screen

`.minibuf.reverse`
: False, Reverse colors for minibuffer screen

`.screen.icon.name`
: "", Icon Name

2.1.2 Command Line Options

Epoch also accepts command line options. These override values in the X defaults. A complete list is in the table below. Note that the program resource name/class can be overridden by specifying different values on the command line (see Section 2.1.1 [X Window Program Defaults], page 5 for how the name and class are used).

`Option` Resource, Effect

`-bg` *background, Sets background colors

`-background`
: *background, Sets background colors

`-fg` *foreground, Sets foreground color

`-foreground`
: *foreground, Sets foreground color

`-geometry`
: *screen.geometry, Sets X-window geometry

`-fn` *font, Sets the font

`-font` *font, Sets the font

`-wn` *name, Sets the X-window title

`-window` *name, Sets the X-window title

`-d` *display, Sets the display to use

`-display` *display, Sets the display to use

`-r` *reverse, Reverses foreground / background colors (no argument)

`-rn` NA, Sets the resource name

`-resource`
: NA, Sets the resource name

`-cn` NA, Sets the resource class

`-class` NA, Sets the resource class

`-motion` *motion, Turns on passing motion events to elisp for all screens

`-xrm` NA, Overrides a resource manager value

2.2 New Features

This section provides a brief overview of the new features in Epoch. More detail is provided in subsequent chapters.

In this document, the term *screen* will mean an X-window, and *window* will refer to an editor window to avoid confusion. In most cases, working within a screen works the same as in GNU Emacs.

Epoch supports two types of screens: *edit screens* and *minibuffer screens*. There is always exactly one minibuffer screen. The number of edit screens can change, but there is always at least one. A single edit screen and the minibuffer screen together act very similarly to normal GNU Emacs.

At any time there is a single *selected screen*, similar to having a selected window. This can also be called the *current screen*. Standard window functions work, but only apply to the current screen. E.g., `one-window-p` returns non-`nil` if there is only one window on the current screen. Buffers are shared among all screens, so any buffer in any window can be accessed, and will show up in buffer lists. Killing buffers removes them from all windows.

Epoch adds *screens* as a new Elisp primitive type. Screens are printed as `#<screen` *n*`>`, where *n* is the *screen ID*. This ID is a sequence number attached to a screen when it is created. It serves primarily as a human convienence. Any argument to a primitive listed as *screen* can be either a screen object or a screen ID. However, when the result of a function is a screen, this means a screen object unless specifically mentioned to do otherwise. Optional screen arguments default to the current screen unless otherwise specified.

Buttons are regions of text in a buffer than can have data attached to them and be displayed in different colors than surrounding text. Buttons provide both the ability to do highlighting and also to attach data to text regions that can be retrieved based on buffer location.

Epoch supports event sending and reception, including client messages and X-window properties, and asynchronous reception of client and property change events. This allows Epoch to communicate with other clients or the window manager.

2.3 Key Bindings

We provide a large range of useful functions to create, delete and change the attributes of screens

(including changing colors, size, etc). We provide a default set of key bindings to these functions, shown below. The basic idea was to take the `C-X` binding for Emacs window and buffer operations, and provide analogs for screen operations using `C-Z` as the prefix key. (We use `C-Z` because a suspend operator is not all that useful in a multi-window environment, and anyway `C-X C-Z` still provides the same effect as the standard.) We do not believe we have a full set of bindings yet, and look forward to user contributions.

We supply a '`.emacs`' and a file '`epoch.el`' that provide the bindings and a good introduction to using the primitive Epoch facilities described in this document. See below for more information.

KEY	Functionality
`C-Z 4 F`	`find-file-other-screen` Find a file in another screen. Prompts for a filename. If it is already in a window on a screen, warp the cursor there, otherwise create a new screen and edit the file in it.
`C-Z 4 B`	`find-buffer-other-screen` Selects a buffer in another screen. Prompts for a buffer name. If it is already in a window on a screen, warp the cursor there, otherwise create a new screen with one window for the buffer.
`C-Z 0`	`remove-screen` Delete current screen.
`C-Z O`	`switch-screen` Circulate through mapped screens in forwards direction.
`C-Z P`	`prev-switch-screen` Circulate through mapped screens in reverse direction.
`C-Z 2`	`duplicate-screen` Duplicate current screen.
`C-Z H`	Hide (unmap) current screen
`C-Z R`	`raise-current-screen` Raise current screen
`C-Z L`	`lower-current-screen` Lower current screen
`C-Z M`	`raise-minibuf` Raise minibuffer screen
`C-Z I`	`iconify-screen` Iconify current screen
`C-Z A`	Force update of title bar for current screen
`C-Z E`	Display event handler status

2.4 Conventions

Several conventions are used in the following chapters.

- Functions and variables are described with the format used in the Emacs Lisp Reference Manual.
- Any *screen* argument can be a screen ID or a screen object. Optional screen arguments default

to the current screen.

- Any *color* argument can be a vector or a string. Strings are converted to vectors internally.

We wanted to keep the necessary modifications to the underlying C substrate of GNU Emacs to a minimum. Therefore, each primitive epoch function exported out of the C substrate has a leading '`epoch::`' in a fashion reminiscent of Common Lisp packages, e.g., `epoch::create-screen`. The point is that we want to build around these functions at the elisp level, so we deliberately make them uncomfortable to use. The file '`epoch.el`' provides a full set of wrappers that remove the '`epoch::`' leading characters and add additional functionality (e.g. there is a function `create-screen` that provides a higher-level interface to the primitives). If you are going to write your own elisp code you should probably use these wrapper functions, not the raw primitives.

2.5 GWM

GWM (generic window manager) is an Emacs-like window manager in that it provides a basic kernel of window-managing facilities written in C and a lisp level for extensive user customization. GWM is part of the X11 release (the most up-to-date version is always available from '`expo.lcs.mit.edu`').

One can attach state machines to objects (such as X-windows) in GWM, and these then receive and handle events. GWM can receive and initiate property change events, and so can Epoch so this provides the basis for communication between the two.

GWM also provides the ability to decorate windows with up to four bars (one on each side, top and bottom). Each bar can have "plugs" (icons) which can have specialized state machines attached. This allows one to have title bars, bars with pulldown menus, etc, all maintained by GWM not by the client, but with communication allowing selections to be communicated.

The effect of this is that we have not provided any widget facilities in Epoch as we believe the desired effect can be gained more efficiently by using the GWM approach.

Fortunately, GWM is easily tailored to emulate other window managers and comes with reasonable emulations of MWM and TWM as well as a "home-grown" look-and-feel, so most users can use it as a replacement for previous managers quite easily.

The interested reader should see the GWM documentation for more details.

3. Screens

In this document, the term *screen* will mean an X window, and *window* will refer to an editor window to avoid confusion. In most cases working within a screen will be the same as in GNU Emacs.

Each screen contains its own Emacs window hierarchy. (This terminology is a little confusing: when GNU Emacs uses the term *window* this refers to the logical structure in which a buffer is displayed. The name "window" was already appropriated, so we had to call our “X-windows” something different. They are therefore called *screens*.) When a screen is first created, it contains a single window covering the entire screen. The buffer for this screen is set the `*scratch*` unless a different buffer is specified.

Epoch supports two types of screens: *edit screens* and *minibuffer screens*. There is always exactly one minibuffer screen. The number of edit screens can change, but there is always at least one. A single edit screen and the minibuffer screen together act much like normal GNU Emacs.

At any time there is a single *selected screen*, analogous to having a selected window, also be called the *current screen*. Standard window functions work, but only apply to the current screen. E.g., `one-window-p` returns non-`nil` if there is only one window on the current screen. Buffers are shared among all screens, so any buffer in any window can be accessed, and will show up in buffer lists. Killing buffers removes them from all windows. The current version of Epoch does not update non-current screens by default (see `epoch::global-update` in See Section 3.5 [Screen Updating], page 22).

Epoch adds *screens* as a new Emacs Lisp primitive type. Screens are printed as `#<screen` *n*`>`, where *n* is the *screen ID*. This ID is a sequence number attached to a screen when it is created. It serves primarily as a human convenience. Any argument to a primitive listed as *screen* can be either a screen object or a screen ID. However, when the result of a function is a screen, this means a screen object unless specifically mentioned to do otherwise. Optional screen arguments default to the current screen unless otherwise specified.

A screen that is displayed is said to be *mapped*; if it is only displayed as an icon, it is said to be *unmapped*. By X Window System conventions, a screen is mapped iff its X window is mapped.

There is one significant difference between screens and windows: screens are known to external programs (e.g., the X-server). This means that the GNU Emacs scheme for window activities is not sufficient for screen activities. In particular, the creation of screens is much more complex than window creation.

epoch::create-screen &optional *buffer alist* Function

Creates a new screen (unmapped), and returns it. If the buffer argument is present, that buffer is attached to the single window originally present in the screen, otherwise the `*scratch*` buffer is used. The *alist* is used to override screen creation defaults.

create-screen &optional *buffer alist* Function

Calls `epoch::create-screen` but does some additional things. The alist passed to `epoch::create-screen` is a copy of *alist* appended with a copy of `epoch-mode-alist` and two addition properties. These additional properties are `icon-name` and `title` (which here default to *buffer name*`@`*system name*).

create-screen-alist-hook Variable

This is a hook variable used by `create-screen`. Just before the call to `epoch::create-screen`, each function in the hook is called with the screen property alist `create-screen` has generated. Each function should return a new version of the alist to be used instead. Normally, the hook function will add or modify some of the entries in the alist. See the code in See Section 7.3 [Screen Pools], page 52 for an example.

3.1 Screen Properties

Screens have a large number of properties (or attributes) that can differ between screens, and are externally visible (e.g., colors, cursor shape and color, font, size, title, class). Most of the complexity of screen creation is involved in dealing with these properties. Epoch deals with *default-sets* which are collections of X window property types and values. When a screen is created, a default-set is used to set the various properties of the screen. Creation of default sets is described in See Section 2.1.1 [X Window Program Defaults], page 5.

In order to allow more variety in screens, Epoch provides more ways of overriding the global X defaults. A global variable `epoch::screen-properties` contains an alist of screen properties. At screen creation time, a copy of the global default-set is made, and entries are overwritten with values from `epoch::screen-properties`. After that, the alist given as an argument to `create-screen` (if any) is used to override any previous values.

The order of determination of X defaults for each screen is as follows (in increasing order of precedence):

- X Window Program Defaults (see Section 2.1.1 [X Window Program Defaults], page 5)

- Values in `epoch::screen-properties`.
- Alist argument to `epoch::create-screen`.
- Mode-specific values using `epoch-mode-alist`, if `create-screen` is used for creation.
- Alist argument to `create-screen`.

epoch::screen-properties Variable

A variable that should contain an alist, the key values being screen attribute keys. The primitives attempt to bypass bad entries, but if the alist is improperly formed, Epoch may experience bizarre failures.

epoch-mode-alist Variable

This variable allows you to associate a set of X defaults with an emacs mode. The form is a list, each element of which is a form (`mode-name` . *alist-of-defaults*). Whenever `create-screen` is called, the mode of the *buffer* argument (or the current buffer's mode if *buffer* is `nil`) is used to look in the alist.

This feature allows you to set geometry, colors, font, etc by mode. Its useful to allows (for example) 80x60 windows for '`.c`' and '`.h`' files, but a regular size window for other file types. See the definition of '`epoch-mode-alist`' in '`epoch.el`' for some examples.

Thus you can set up a default mode upon entry and then override portions of it at the time you call `create-screen`.

The table below is a list of all screen property keys recognized by the screen creation routines. Defaults are the hard-wired defaults. X defaults and command line options can override them. Values listed as `flag` are false if the value is `nil`, true for any other value.

As an example, to set screens to have the font '`9x15`' and the '`Gumby`' cursor, the following Elisp code would be used:

```
(setq epoch::screen-properties
  (cons '((font . "9x15") (cursor-glyph . 56))
        epoch::screen-properties))
```

`Name` Type, Default, Description

`foreground`
String, Black, Screen foreground

`background`
: String, White, Screen background

`cursor-color`
: String, Black, Text cursor color

`cursor-foreground`
: String, White, cursor foreground color

`cursor-background`
: String, Black, cursor background color

`border-color`
: String, White, X border color

`in-border-width`
: Number, 2, Internal border width

`ex-border-width`
: Number, 1, External border width

`title` String, "Edit", Screen title

`name` String, "epoch", Screen resource name

`class` String, "Emacs", Screen class name

`update` String, false, Flag to indicate non-local screen updating

`geometry` String, 80x24, Geometry specifier. Size units are characters. Screen must be at least 2 lines high.

`cursor-glyph`
: Number, 86, X cursor glyph. Index into the cursor font.

`reverse` Flag, False, Reverse foreground / background

`font` String, "fixed", Font name

`icon-name`
: String, "", Name of Icon for Screen

`motion` Flag, nil, `t` will cause motion-events to be sent to screen, `nil` will inhibit this.

`parent` X resource, nil, If non-`nil`, must be an X "resource id". The new screen is created as a child of this resource. Useful to provide clipping-parents, for example when trying to put screen in a form of some kind. *This option is extremely dangerous and you should not use it if you dont know what you are doing.*

`initial-state`
: Flag, True, Starting state of screen. True means NormalState, false is IconicState.

3.2 Controlling Screens

The primitives in this section concern internal manipulations of screens.

epoch::get-screen &optional *screen* Function

Coerces a *screen* or screen ID argument to the corresponding screen object.

epoch::get-screen-id &optional *screen* Function

Coerces a *screen* or screen ID to the corresponding screen ID. Returns `nil` if the screen is deleted.

make-screen &optional *screen* Function

Use get-screen instead. OBSOLETE.

make-id &optional *screen* Function

Use get-screen-id instead. OBSOLETE.

epoch::screen-p *screen* Function

Returns `t` if the argument is of type *screen*, `nil` otherwise. This function should be called `epoch::screenp`.

epoch::screen-list &optional *unmapped* Function

Returns a list of all mapped screens, except the minibuffer which is never in the list. If the argument *unmapped* is present and non-`nil` then unmapped screens are also included.

epoch::next-screen &optional *screen unmapped* Function

Returns the next screen in the internal ordering. With no arguments, it is the next past the current screen. A *screen* argument causes the return of the next screen past the argument. Unmapped screens are not returned, unless the second argument *unmapped* is present and non-`nil`. The minibuffer screen is never returned.

epoch::prev-screen &optional *screen unmapped* Function

As `epoch::next-screen` except the internal screen list is traversed in the opposite order.

epoch::select-screen &optional *screen* Function

Makes the argument *screen* the current edit screen. The default is the next screen, as defined by `epoch::next-screen`. The minibuffer screen cannot be selected. Unmapped screens may be selected, but only explicitly or if only unmapped screens are left. The

selected window is changed to be one in the new screen. Epoch attempts to remember what window was last selected in each screen, and selects that one.

Returns the selected screen, or `nil` if given a bad argument or if *screen* has been deleted.

Note: If a window is selected by `select-window`, an implicit `epoch::select-screen` is done to select the screen the window is on. This insures that the current window is on the current screen whether `epoch::select-screen` or `select-window` was used. In fact, this primitive actually works by selecting a window in the new screen.

epoch::current-screen Function

Returns the current edit screen. The minibuffer screen is never returned.

epoch::minibuf-screen Function

Returns the minibuffer screen.

epoch::delete-screen &optional *screen* Function

Deletes a screen. Returns `t` if successful, `nil` otherwise. It is an error to delete the sole remaining screen.

If you delete a screen and it is the current screen, it uses `epoch::select-screen` to select a new screen.

BUG: Window managers can delete the last edit screen. If this happens, Epoch will crash and burn in short order. Window managers *should* obey the ICCCM standard and send a delete signal to the client; then Epoch can grab this and terminate gracefully.

3.3 Screens and Windows

Epoch supports all of the windowing ability of GNU-Emacs on each screen. Every window is on a screen, and can never be moved from that screen. Epoch provides some primitives to help in the interaction between screens and windows.

epoch::get-buffer-window *buffer* Function

Searchs for a window displaying *buffer* on all screens. Returns such a window if successful, `nil` otherwise. This is effectively an updated version of the normal primitive

`get-buffer-window`, which searches only the current screen and minibuffer.

Note: selecting a window also selects the screen it is on. Remember that the window may be on an unmapped screen. This function has no wrapper since that would collide with the standard function `get-buffer-window`.

epoch::screens-of-buffer *buffer* Function

Returns a list of all screens on which *buffer* is displayed in a window.

epoch::screen-of-window *window* Function

Returns the screen that *window* is on.

epoch::first-window &optional *screen* Function

Returns the first window in canonical order for *screen*.

epoch::selected-window &optional *screen* Function

Returns the selected window for *screen*. This is the window that would be selected if the screen it is on were selected, or equivalently the selected window the last time *screen* was the current screen.

Note: There is no wrapper for this function, since that would collide with the standard function `selected-window`.

3.4 Manipulating Screens

These primitives allow control of the windowing system aspects of screens. Most of them correspond directly to X-window calls.

Each of these functions return the screen raised, if successful, or `nil` otherwise.

The *screen-or-xwin* arguments may be screen objects or X resource object with type window (see Section 6.1 [X Resources], page 45).

epoch::raise-screen &optional *screen-or-xwin* Function

Raises the *screen-or-xwin* to the top of the display.

epoch::lower-screen &optional *screen-or-xwin* Function

Lowers the *screen-or-xwin* to the bottom of the display.

epoch::map-screen &optional *screen-or-xwin* Function

Maps the *screen-or-xwin* onto the display. No effect if the screen is already mapped.

epoch::unmap-screen &optional *screen-or-xwin* Function

Unmaps the *screen-or-xwin*. This is for sophisticated users only, as it does not check to see if the screen is the edit screen or the minibuffer. If the unmapped screen is the edit screen, it remains so - you just won't be able to see it.

epoch::mapraised-screen &optional *screen-or-xwin* Function

This does a MapRaised call on the *screen-or-xwin*. This is equivalent to doing a map followed atomically by a raise. If no argument is present, it uses the current screen.

epoch::iconify-screen &optional *screen* Function

Sends ICCCM client message requesting iconification of screen (or current screen if no argument present).

Note:

- Some window managers are not ICCCM compliant and will not handle this correctly.
- ICCCM standard says that to uniconify, an `epoch::map-screen` should be used. This doesn't seem to work for many window managers at present.

epoch::screen-mapped-p &optional *screen* Function

Returns `t` if the *screen* exists and is mapped, `nil` otherwise.

epoch::screen-height &optional *screen* Function

Returns the height (in lines) of the *screen*. A generalization of `screen-height`.

epoch::screen-width &optional *screen* Function

Returns the width (in columns) of the *screen*. A generalization of `screen-width`.

epoch::change-screen-size *width height* &optional *screen* Function

Attempts to resize the *screen*. The *width* and *height* are in characters. This uses the `XResizeWindow()` call, which may be intercepted by the window manager. The *width* and *height* values are adjusted be be not more than 150. The *height* is adjusted to at least 1 for the minibuffer and 2 for edit screens.

epoch::font &optional *font screen* Function

If called with `nil` *font* argument, this function returns the current font. If called with `font` argument, it changes *screen*'s font to be that font. It will resize the screen so that it still has the same character geometry. It operates on the current screen if *screen* is `nil`. The result is a list of three elements: Font name, character-width, character-height in pixels.

epoch::plane-size Function

Returns a cons pair. Car is plane width, cdr is plane height in pixels. (A plane is a "minor screen" on a display: on mono screens, there is just one plane, color screens sometimes have several).

epoch::screen-information &optional *screen-or-xwin* Function

Returns information about *screen-or-xwin*, or the current screen if *screen-or-xwin* is `nil`. The information is a list with the following elements:

- X location of screen
- Y location of screen
- Width of screen
- Height of screen
- External Borderwidth
- Internal Borderwidth; for an xwin, this is always 0.
- Map state: `t` if mapped, `nil` if not.

epoch::move-screen *x y* &optional *screen-or-xwin* Function

Sends request to window manager to move upper-left corner of *screen-or-xwin* (or current screen if no third argument) to given (*x*,*y*) coordinates. Note that some window managers may choose to ignore this request.

epoch::foreground &optional *color screen* Function

Set/get the foreground color of the screen. If *color* is `nil` then the function returns

the current foreground color triplet. Otherwise it attempts to set the foreground color, and return `t` on success and `nil` otherwise.

epoch::background &optional *color screen* Function

Set/get the background color of the screen. If *color* is `nil` then the function returns the current background color triplet. Otherwise it attempts to set the background color, and return `t` on success and `nil` otherwise.

epoch::cursor-color &optional *color screen* Function

Set/get the text cursor color of the screen. If *color* is `nil` then the function returns the current text cursor color triplet. Otherwise it attempts to set the text cursor color, and return `t` on success and `nil` otherwise.

epoch::cursor-glyph &optional *glyph screen* Function

Set/get the X cursor glyph. If *glyph* is `nil`, returns the current glyph number, otherwise attempts to set the glyph.

epoch::flash-screen &optional *screen-or-xwin* Function

Flash the *screen-or-xwin*. If the argument is a screen object, the flashing is done by exchanging the foreground and background colors, pausing .25 seconds, and then exchanging them again. For an X-window resource (even if it is an Epoch screen), the flashing is done by inverting the pixels, pausing, and then inverting again. This is effective on a monochrome display, but is not as effective on for color.

3.5 Screen Updating

Epoch has a feature that allows you to control which screens are updated as their contents are changed. The default is for only the current screen to update, which is maximally efficient. It is possible, however, to set things up so that all screens or only some subset of the screens are updated.

This is useful if you are editing a buffer which is displayed on multiple screens, or if you have a shell or other process running in one buffer and you are working in another. Then you can set things up so that the screen with the shell continues to update as it runs, while you work in another screen. There are several ways to do this, all involving the value of the global variable `epoch::global-update`.

The functions and variables are:

epoch::global-update Variable

There are three classes of values for this variable.

- set to `nil`. Disables updating all but the local screen.
- set to `t`. Enables potential updates: particular screens to be updated conditionally based on screen-local `epoch::set-update` functions or through the default property alist value `update`, or on the `epoch-mode-alist`, or on the property alist.
- set to any other value: causes all screens to be updated. This is a shorthand for setting to `t` and then setting the update property on each screen individually.

epoch::set-update *arg* &optional *screen* Function

Sets update flag to *arg* for *screen*, or current screen if nil. *arg* must be `t` for update, `nil` for no update. Returns previous value of flag.

epoch::update-p &optional *screen* Function

Returns `t` if *screen*'s update flag is set, `nil` otherwise. Uses current screen if *screen* argument is not present. This function should be `epoch::updatep`.

In addition you can use the following primitives to help control when screens get updated:

epoch::redisplay-screen &optional *screen* Function

Forces a non-premptive redraw of argument *screen* (current screen if no argument is supplied.

epoch::set-screen-modified &optional *screen* Function

Mark the argument *screen* (current if omitted) as modified for update next time Epoch gets around to an update.

4. Buttons

Epoch supports the notion of *buttons*, which are regions that are displayed in a different foreground/background color combination. A new data type, a button, has been added to Emacs. Each button has the following characteristics: *start*, *end*, *attribute*, *data*, and *read-only*. *start* and *end* are positions delimiting the button, and are stored as integers, although they behave just like markers. The *data* is a user defined field, which is not used by Epoch itself.

The *attribute* is an index into a table of *button attributes*. If a button has an attribute which has not been specified, inverse (highlight) style is used. Because normal cursor colors are used for the cursor, buttons on a monochrome screen will hide the cursor unless the attribute is explicitly set.

Buttons were intended as a way of providing "hotbutton" capabilities in hypertext-like applications using Epoch as a front-end. In these cases the data field will hold information for the hotbutton. Buttons can also be used for many other applications, of course.

Each button is attached to a buffer. All buffer arguments in the Epoch primitives default to the current buffer. Arguments of type button must button objects.

Functions that return buttons or button lists do not return the actual objects. Functions exist to access and change a buttons start/end position or even the button's buffer, while maintaining the correct position in the button list. **NOTE:** Elisp code that alters buttons by changing the vectors directly will no longer work correctly.

The wrapper functions and the non-primitive functions described here are provided in '`button.el`', not in '`epoch.el`'.

4.1 Button Primitives

epoch::make-button Function

Used to create a new button. Returns a button pointing nowhere, in no buffer.

epoch::buttonp *object* Function

Returns `t` if object is a button, otherwise `nil`

epoch::button-buffer *button* Function

Returns the buffer a button is currently in, or `nil`.

epoch::button-start *button* Function

Returns the button's *starting* position as an integer.

epoch::button-end *button* Function

Returns the button's *end* position as an integer

epoch::move-button *button start end* &optional *buffer* Function

Moves the specified button to a new position indicated by *start* and *end* in *buffer*. If either paramter is `nil`, it will remain unchanged. Moving buttons will cause the respective buffer(s) to redisplay, but will **not** mark the buffer as modified unless `epoch::buttons-modify-buffer` is `t`. Returns the updated button.

start and *end* may be markers, but only the marker position is used .

epoch::add-button *start end attribute* &optional *data buffer* Function

Add a button to *buffer*. *start* and *end* must be markers or integers. *attribute* is a number. *data* is programmer defined, and not used internally, and so can be anything. Returns the new button, or `nil` on failure. (This function is provided for compatibility with the Epoch 3.2 release.)

epoch::button-at &optional *position buffer* Function

Returns the button in *buffer* that contains *position*, or `nil`. If *position* is `nil`, point is used.

Epoch needs a `epoch::buttons-at` function to get all butttons.

epoch::button-list &optional *buffer* Function

Returns the list of buttons for *buffer*. The list contains the actual buttons, not copies.

epoch::button-data *button* Function

Returns the button's current *data* value.

epoch::set-button-data *button value* Function

Sets the button's *data* to *value*.

button-text &optional *button* Function

Returns a string that is the text contained in *button*. If *button* is omitted, the `button-at` point is used, if any. Returns the empty string for an empty or no button.

epoch::buttons-modify-buffer Variable

If non-`nil`, changing any buttons with Epoch primitives will cause the associated buffer to be marked as changed. Useful if you want `write-file-hooks` that save button information.

4.2 Deleting Buttons

epoch::delete-button *button* &optional *buffer* Function

Removes the *button* from *buffer*. The argument must be the actual button itself, not a copy. This can be obtained by saving the result from `epoch::move-button` or other button returning primitives. Returns the button if successful, `nil` otherwise.

epoch::delete-button-at *position* &optional *buffer* Function

Removes the button (if any) that contains *position* in *buffer*. Returns the button if successful, `nil` otherwise. Equivalent to `(epoch::delete-button (epoch::button-at position buffer) buffer)`.

epoch::clear-buttons &optional *buffer* Function

Removes all buttons from *buffer*. Returns `t` if succesful, `nil` otherwise.

4.3 Button Attributes

An attribute is a pair of foreground / background colors, one set for text and the other for the cursor. Each screen has a set of 256 attributes, of which 255 are user selectable. Attribute 0 is always normal text and cursor. Attributes in buttons that have not been set use reverse (highlight) for plotting text, and normal cursor colors for the cursor.

WARNING: This means that on a monochrome screen, if buttons are used with unset attributes, the cursor will be invisible in the button.

Cursor colors default to be the inverse of the text color for the attribute, although they can be specifically set.

Each screen can be set to have either *local attributes* or *global attributes*. All screens with global attributes can have attributes set with a single primitive. Local attributes must be updated directly. Screens can be switched from local to global through other primitives. When changed from local to global, the current set of global attributes is copied onto the screen attributes. Changing to local affects only future global attribute setting. Current attributes are unchanged. It is possible to use local attribute setting on global screens. In this case, only the attribute on that screen is changed, even though the screen has global attributes. However, if the same attribute is later set globally, the attribute on the screen will be updated at that time. By allocating ranges of attributes for global and local use, screens can maintain a standard set of attributes while having local variation.

When a buffer is plotted, buttons are extracted from the buffer button list and used to change the colors of the text and cursor. If the buttons are disjoint, then which button to use for a character is unambiguous. Overlapping buttons are more complex; the algorithm for deciding which attributes are used when buttons overlap is described next.

The button plotter assumes that the button list is sorted by starting position. If only button editing primitives are used, this is always true. If buttons overlap, the button with the largest starting position takes precedence. This allows buttons to nest, so that if one button is entirely inside another, both buttons are visible. If two buttons overlap without containment, the second button will be entirely visible, and the first truncated. Whenever there is a reference to the button "at" a location, that means the button that was plotted at that location.

A button with a non-zero attribute always has precedence over a button with zero attribute. This means that if a button with an attribute of zero follows and overlaps a button of non-zero attribute, the preceeding button will be plotted in the overlap region. Zero attributes are intended for hidden or non-user buttons (such as for tagging a region of text with data), and this rule prevents them from interfering with visible buttons.

epoch::button-attribute *button* Function

Returns the button's current *attribute* value.

epoch::set-button-attribute *button value* Function

Sets the button's *attribute* to *value*. Note that *value* must be a number less than 255.

epoch::set-attribute-global *number fore back* &optional *c-fore c-back* Function

Sets the global attribute *number* to foreground *fore* and background *back*. The cursor colors are set to *c-fore* and *c-back*. If these are not specified (or are `nil`), then the opposite text color is used. All of *fore*, *back*, *c-fore*, and *c-back* should be colors. Sets the global attribute, and then copies the colors into all screens that have global attributes. Returns `t` if successful, `nil` otherwise.

epoch::set-attribute-local *number fore back* &optional *c-fore c-back* *screen* Function

Similar to `epoch::set-attribute-global`, except that only the attribute on *screen* is set, regardless of whether the screen has local or global attributes.

epoch::local-buttons-p &optional *screen* Function

Returns `t` if the screen has local button attributes, `nil` if global.

epoch::set-local-buttons *flag* &optional *screen* Function

If *flag* is `nil`, the screen is set to have global attributes, and current values of the global attributes are copied into the screen attributes. If non-`nil`, the screen is set to have local attributes, but no change to current attributes occurs. Returns the previous local button attributes state.

4.4 Attribute Allocation

In order to use buttons, specific attributes must be set by the button code. To avoid collisions between different packages, an attribute allocator is provided in '`button.el`'. This allows packages to reserve attributes for their own use, and if desired release them to be reused. It is strongly recommended that if you write a package that uses buttons, you use the attribute allocator.

reserve-attribute Function

Returns a number, the index of the allocated attribute.

reserve-attributes *count* &optional *noerror* Function

Reserves *count* attributes, and returns the list of corresponding attribute indices. If *noerror* is non-`nil`, and not enough attributes are available, a short list is returned, otherwise an error is signaled and no attributes are marked reserved. If *count* is omitted, is as assumed to be 1.

release-attribute *attr* Function

Mark *attr* (which should be an attribute index) as available.

release-attributes *attr-list* Function

Mark all attributes in *attr-list* as available. *attr-list* should be a list of attribute indices.

4.5 Read-only Buttons

In the case of a *read-only button*, (the *read-only* field is `t`), most attempts to change text within the button will result in an error analogous to changing a read-only buffer. Two exceptions to this rule exist at button boundaries.

1. Attempted changes on the first character of the button will be allowed, but will be diverted to the left of the button itself.
2. Attempted changes to the immediate right of the button will be permitted, but the button will not be extended.

Essentially, this implies that the size of a read-only button will only change because of explicit commands; **not** by inserting or deleting text within the button. Changing the location of a read-only button by altering text previous to the button is permitted; this is even possible with adjoining read-only buttons (unless they explicitly overlap).

add-read-only-button *start end attribute* &optional *data buffer* Function

The same as `epoch::add-button` except that the button is set to be read-only immediately after being created.

epoch::read-only-region-p *start end* &optional *buffer* Function

Given a region delimited by *start* and *end* in *buffer*, check for read-only buttons. Returns `nil` if region is clear, otherwise returns the button. In cases of overlapping buttons, the display rule discussed earlier is obeyed (see discussion of button plotting in See Section 4.3 [Button Attributes], page 27). If any portion of the region is in any read-only button, it returns a button.

Note: When checking for insert operation, use the insertion point for both start and end. If this would normally be inside the button, but because of insertion rules would be deflected to the side of the button, this function will not return the button.

epoch::button-read-only *button* Function

Return's the button's current *read-only* value.

epoch::set-button-read-only *button flag* Function

Sets the button's *read-only* flag to *flag*, which is either `t` or `nil`.

5. Events

Epoch provides the ability to handle X window system events in Elisp.

5.1 Event Basics

Events are the mechanism the X window system uses to communicate with client programs (such as Epoch). Certain types of X events are stored into a common queue, the *event-queue*. These types are listed below. Whenever such an event is received by Epoch, it is stored in the queue, and an "event-key" is inserted into the keyboard buffer. Whenever the keyboard is checked for keys, these "event-keys" are read until there none are left or an actual key is encountered. For each of these events, the variable `epoch::event` is bound to the value of the event, and the event handler `epoch::event-handler` is called. (See Section 5.2 [Event Handling], page 34 to learn more about that subject.) Each event is returned as a vector of 3 elements. The first is the event type, the second is data associated with the event, and the third is the screen on which the event occured.

Epoch currently captures the events described in the following table:

`Name`	Data
`focus`	`t` for focus-in, `nil` for focus-out
`property-change`	If `epoch::lazy-events` is `nil`, then a cons pair. Car is name of property (a string), cdr is new value of property. Otherwise, the value is just the X-atom resource of the property.
`map`	`t` for map, `nil` for unmap
`move`	Cons pair: x-position, y-position
`resize`	List of 3 elements: new width, new height, outer borderwidth
`client-message`	A triple of form (*TYPE FROM . DATA*). See description of `epoch::send-client-message` below for an explanation.
`button`	A 5-tuple of form (*Press/Release x-coord y-coord button mod-state*)
`motion`	A triplet of the form (*x-coord y-coord mod-state*).

To support asynchronous communication through events, Epoch provides primitives to set and read properties, and a property event handler. Primitives are also provided to send client messages.

Properties are manipulated and accessed through the `epoch::set-property` and `epoch::get-property` functions defined below.

Client messages are sent through the `epoch::send-client-message` function and received through events of type `client`.

Epoch maintains two properties on the screens automatically. These are `EPOCH_SCREEN_ID`, which is an integer that is the screen ID, and `EPOCH_CURRENT`, which is a string, either "yes" if that screen is the current edit screen, "no" if it is not, or "minibuf" if it is the minibuffer.

One way that property changes can be used is to send messages to and receive messages from the window manager. We use this facility extensively to display menus to the user, change window names and decorations, etc.

epoch::get-event Function

All events are stored in an internal queue. This primitive returns the oldest event in the event queue as an Elisp vector of size three. The zeroth element is the type of the event, the first element is the value of the event, and the second element is the screen on which the event occurred. **OBSOLETE**

epoch::lazy-events Variable

If this variable is non-`nil`, then the internal event code will do as little work as possible. Primarily, property events will only report the atom of the property changed, and leave it up to other code to acquire the value if desired.

epoch::mouse-events Variable

If this variable is non-`nil`, then mouse events are placed into the Epoch event queue instead of generating fake key strokes.

5.2 Event Handling

Events are handled by binding the variable `epoch::event-handler` to a function of no arguments. Whenever an event is dispatched as described above, if `epoch::event-handler` is non-`nil`, the value in it is called as a no argument function. This function should then examine the value of `epoch::event` and take appropriate action.

If any error occurs while the event handler is being executed, then the behaviour depends on the value of the variable `epoch::event-handler-abort`. If this is `t`, then the `event-handler` is aborted by resetting the variable to `nil`. This action prevents runaway errors on every following event. If the variable `epoch::event-handler-abort` is `nil` then this action is not taken.

In order to make event handling easier, a standard set of event handling functions are provided. These provide a (somewhat) widget-like interface to events. Selected parts of the interface can be replaced without change to the underlying internals. Event handling is organized hierarchially – the top level handler invokes other handlers to dispatch the final handling routines. The following description is from the top down.

The basic event handler maintains an internal alist to associate event types with event handlers. An *event type* is the symbolic name that X uses in event handling. An *event handler* is an elisp function that may be called when an event occurs.

For each event type, a stack of event handlers is kept, and when an event of that type occurs, the function at the top of the stack is dispatched. This allows you to change the event handler behaviour in a restorable way. Each basic event type must be installed before any events will be dispatched for it; this allows events to be globally disabled and reenabled. Currently the Epoch elisp files (see Section 7.2 [Packages], page 49) install all supported events.

When the basic handler runs, `epoch::event-handler-abort` is set to `t`. It then watches for errors during the dispatch of an event, which are detected by checking `epoch::event-handler` after returning from the dispatch. If `epoch::event-handler` has been set to `nil`, an error is presumed to have occured, and `ignore-event` is called on that event. This provides improvements over the built-in system, in that only a particular type of event is ignored (others continue to dispatch), and service can be restored with a simple (`resume-event` *event*), independent of the actual names of the handlers. This means that handlers can now be anonymous, i.e. `push-event` can be passed a `lambda` expression instead of the symbol name of a function.

Event handler functions should be defined as follows:

```
(defun my-event-handler (event-type value screen)
  "event-type is the type of the event.  value is the value of
the event, and screen is the screen which received the event.
Handlers can call the event handling interface routines, so that a
one-shot handler (which pops itself) can be set up."
  )
```

The following functions and variables form the interface to the event handling system.

epoch::event-handler Variable

Variable that contains `nil` or a function to be called when there are pending property change events. The handler should use the variable `epoch::event` to retrieve the current event.

epoch::event Variable

Just before the event handler in `epoch::event-handler` is called, this variable is bound to the value of the event that caused the handler call. It is a vector of size three, consisting of the event type, the event value, and the screen where the event occurred.

epoch::event-handler-abort Variable

If this variable is non-`nil`, then if an error occurs while the event handler is running, `epoch::event-handler` is set to `nil`. This prevents a error in some part of the event handler code from causing Epoch to lock up with excessive errors.

install-event *event* Function

This puts *event* into the handler alist. *event* can be anything, but unless it is `eq` to an event type, it will never be dispatched.

remove-event *event* Function

Removes *event* from the handler alist. All handlers associated with it are lost.

push-event *event handler* Function

Installs *handler* for *event*. Handlers are installed in a stack, and the handler at the top of the stack is dispatched for the event.

pop-event *event* Function

Removes and returns the top level handler for *event*.

ignore-event *event* Function

Pushes a marker onto the handler stack for *event* so that the event is not dispatched.

resume-event *event* Function

Pops the ignore marker from the top of the handler stack for *event*. Note that these two functions are different from `remove-event` and `install-event` in that these allow temporary, transparent inhibition of events in a local area, while the latter have global effects.

display-event-status Function

Print the current event handler status into a buffer and display it in a window on the current screen. Normally bound to `C-z e`. The status keys are `H` for being handled, `I` for ignored, and `U` for uninstalled. If a number follows the letter, that is the depth of the handlers or ignore markers.

5.3 Property Events

The Epoch elisp files set up a handler for the `property-change` event which then dispatches based on the property, in a manner analagous to the basic event handling. This property event handler uses *lazy events* (see `epoch::lazy-events` in See Section 5.2 [Event Handling], page 34). This means that a property change event has a value of just the X-atom resource corresponding to the property that changed. Each installed property is stored as the X-atom resource and a stack of handlers. If the top entry is a function, then it is called, with the same parameters as for basic event handling. (The type will always be `property-change`, but this allows a consistent handler interface for all handler functions.) If the property is not in the alist, or the top entry is not a function, the property change is ignored.

Property change handler functions should be of the same form as the general event handlers. Note that the value of the property change is passed as an X-atom. If the value of the property is desired, the function `get-property` can be used. If the name of the property is needed, use `unintern-atom`. Events are passed this way to save the expense of unneeded round-trip server requests. With non-lazy events, all property events involve 2 round-trip requests, one for the atom and another for the value. However, most property events are not handled, and so such information is simply thrown away. With this system, only those property events that are actually dispatched invoke this expense, and in many cases the name of the property is unecessary. (If the handler function is attached to only one property, then it implicitly knows the property name value.) X-atoms can be directly compared with `equal` so that it is unecessary to convert to strings for comparison.

In contrast to basic event handling, if the property is not installed, pushing the property handler will install it. The inhibition of handling for certain properties is not as useful as for events, while the number of possible property events is far larger than the number of possible event types, the extra hassle of having to seperately install properties was judged to be too much. The `install-property` and `remove-property` will be used seldom, if at all, and are provided primarily for conformity with basic event handling.

The following functions are the interface to the property event handling system.

install-property *property* Function

This puts *property* into the handler alist.

remove-property *property* Function

Removes *property* from the handler alist. All handlers associated with it are lost.

push-property *property handler* Function

Installs *handler* for *property*. Handlers are installed in a stack, and the handler at the top of the stack is dispatched for the property. If the property is not already in the handler list, it is added.

pop-property *property* Function

Removes and returns the top level handler for *property*.

ignore-property *property* Function

Pushes a marker onto the handler stack for *property* so that the property is not dispatched.

resume-property *property* Function

Pops the ignore marker from the top of the handler stack for *property*.

5.4 Client Message Events

The Epoch elisp files set up a handler for the `client-message` event which then dispatches based on the message type. Each installed property is stored as the X-atom resource and a stack of handlers. If the top entry is a function, then it is called, with the same parameters as for basic event handling. If the message type is not in the alist, or the top entry is not a function, the message is ignored.

The value of a client message is a list of (*type source* . *data*) where *type* is the message type X-atom resource, *source* is an X-window resource specifying the source window, and *data* is the data field of the message converted to Elisp data in the same manner as X property values (see Section 6.2.2 [X Properties], page 47). To send client mesages, see Section 6.2.1 [Client Messages], page 47.

If the message type is not installed, pushing a handler will install it. `install-message` and `remove-message` will be used seldom, if at all, and are provided primarily for conformity with basic event handling.

The following functions are the interface to the client message event handling system.

install-message *message* Function

This puts *message* into the handler alist.

remove-message *message* Function

Removes *message* from the handler alist. All handlers associated with it are lost.

push-message *message handler* Function

Installs *handler* for *message*. Handlers are installed in a stack, and the handler at the top of the stack is dispatched for the message. If the message is not already in the handler list, it is added.

pop-message *message* Function

Removes and returns the top level handler for *message*.

ignore-message *message* Function

Pushes a marker onto the handler stack for *message* so that the message is not dispatched.

resume-message *message* Function

Pops the ignore marker from the top of the handler stack for *message*.

5.5 Mouse Events

Mouse events go into the event queue with event type `button` and data the 5-tuple (*press/release x-coord y-coord button mod-state*). *press/release* indicates if the event was a button press or release; the coordinates are character coordinates within the screen; *button* indicates which button; and the *mod-state* is a bitmask indicating the state of the buttons.

Note: Mouse events are only received on the keyboard queue when `epoch::mouse-events` is `nil`. Currently this is the default value, but the Epoch Elisp files set it to `t`.

The Epoch Elisp files install a handler for the `button` events, which in turn dispatches functions for each different mouse event.

Mouse event dispatch uses two tables, one global and the other local. When a mouse event is dispatched, the local table is checked first. If the table is missing or the entry is `nil`, then the global table is used. If a function is found, then it is dispatched. The mouse event handler should be defined as follows:

```
(defun handler (mouse-data)
  "mouse-data is a list of ( point, buffer,
window, screen ).  These all refer to the character location
at which the button on the mouse was pressed or released.  Note that at
the time the handler is called,  no change to point, current buffer,
current window or current screen has been made.  It is entirely up to the
handler how much to change such information.
  )
```

The following functions make up the interface to the mouse handler:

When specifiying a particular mouse event, both the mouse button and keyboard modifier states must be specified, e.g. the middle button with Control. These values are specified as numbers, but for ease of programming constants are defined for them.

These are predefined constants for specifying the mouse button:

`mouse-left`

`mouse-middle`

`mouse-right`

These are the predefined constants for specifying the keyboard modifier state. Names without a trailing `-up` signify the down button event.

`mouse-down`

`mouse-up`

`mouse-shift`

`mouse-shift-up`

`mouse-control`

`mouse-control-up`

`mouse-control-shift`

`mouse-control-shift-up`

`mouse-meta`

`mouse-meta-up`

`mouse-meta-shift`

`mouse-meta-shift-up`

`mouse-meta-control`

`mouse-meta-control-up`

`mouse-meta-control-shift`

`mouse-meta-control-shift-up`

Note: This version of the mouse handling code is very different from the previous version, whose main advantage was that it was the same as regular emacs. In that version, the passed argument was a list of the form (*X Y screen*) of the mouse event. This data was not particularly useful, and had to be converted into more useful data by actually setting point with the `x-mouse-set-point` call. A look through the mouse handling code indicated that in fact every function went ahead and did this in order to calculate the point.

The following functions are modeled after keymaps.

create-mouse-map &optional *source-map* Function

Creates a mouse map for use in dispatching. If *source-map* is present, then the contents of that map are copied into the new map.

copy-mouse-map *from to* Function

Copies a mouse map from *from* to *to*.

use-local-mouse-map *map* &optional *buffer* Function

Sets *map* to be the local mouse map in *buffer*.

kill-local-mouse-map &optional *buffer* Function

Removes the local mouse map from *buffer*.

define-mouse *map button modifier function* Function

Sets the entry in *map* for the mouse *button* with *modifier* to be *function*. *button* should be one of the mouse button constants, and *modifier* should be one of the button modifiers.

local-set-mouse *button modifier function* Function

Sets the entry in the local mouse map for *button* and *modifier* to *function*.

global-set-mouse *button modifier function* Function

Sets the entry in the global mouse map for *button* and *modifier* to *function*.

mouse::set-point *mouse-data* Function

Sets the point to the value specified in *mouse-data*, which should be the same list as passed to mouse handler functions.

epoch::coords-to-point *x y screen* Function

Converts a character *x*, *y* location on a given *screen* into a list of the form `(point buffer window screen)` or `nil` if the location is not on the *screen*.

mouse::convert-xy-screen *xy screen* Function

Converts an (X Y) character position list *xy* and a *screen* into a list of `(point buffer window screen)` without changing the current buffer, window, or screen. OBSOLETE - use `epoch::coords-to-point`

5.6 Motion Events

Epoch allows the user to receive motion events. By default this feature is turned off. It can be enabled in a number of ways:

- using command line option `-motion`: makes motion events appear for all screens.
- using `*motion` in resources: makes motion events appear for all screens.
- using `motion` in the property alist; `t` turns on motion events at screen creation time, `nil` disables them.
- using primitive `epoch::set-motion-hints`: turns motion events on or off by screen.

Note that motion events go into the queue but must be "requested". Each time you execute a function affecting the mouse, for example `epoch::query-pointer`, the next motion event (if any) is sent to the user. This prevents motion events coming so fast that you thrash to death.

The relevent primitives are:

epoch::motion-hints-p &optional *screen* Function

Returns `t` or `nil` for *screen* (default: current screen) depending if motion events are enabled or disabled for that screen.

epoch::set-motion-hints *flag* &optional *screen* Function

Flag must be `t` or `nil`; enables or disables motion hints for *screen* (default: current screen).

5.7 On Event Handling

As part of the standard event handling, a facility to install one-shot handlers for events on particular screens is provided. After the action function has been called for the particular event, it is removed.

on-map-do *screen action* Function

On the next map event for *screen*, call *action* with a single argument that is the screen of the event.

on-unmap-do *screen action* Function

On the next unmap event for *screen*, call *action* with a single argument that is the screen of the event.

on-move-do *screen action* Function

On the next move event for *screen*, call *action* with 2 arguments, the screen and the move event value.

on-resize-do *screen action* Function

On the next resize event for *screen*, call *action* with 2 arguments, the screen and the resize event value.

6. X11 Primitives

6.1 X Resources

Various primitive X11 facilities are provided which we describe here. These give the ability to work with raw "X resources", such as the X server's internal ID numbers for screens (X "windows"). These numbers are too big to fit into the standard GNU Emacs number size, so they are made a special opaque type called an *X-resource*. X-resources contain X-window id information (32 bit quantity), and a type, which is always an X-window system atom. When the type of an X-resource is a predefined value (such as "atom"), then it is referred with that type, (e.g. X-atom resource). Currently recognized types are `Arc`, `Atom`, `Bitmap`, `Cardinal`, `Cursor`, `Drawable`, `Font`, `Integer`, `Pixmap`, `Point`, `Rectangle`, `String`, `Window`, `WM Hints`, `WM Size Hints` or `Resource` (untyped).

epoch::intern-atom *name* Function

Interns *name* as an X-atom resource (as opposed to Elisp atom). X-atom resources are 32-bit numbers that represent strings in the X-window system. They are used for typing information on X-window data objects.

epoch::unintern-atom *name* Function

Converts the X-atom resource *name* to an Elisp string.

epoch::resourcep *arg* Function

Returns `t` if *arg* is an X resource, `nil` otherwise.

In addition, the primitive `equal` has been extended to accept x resources. Two resources are `equal` if their values are the same, regardless of type.

epoch::string-to-resource *string type* Function

Converts an elisp *string* to an X resource (raw window id, for example) of given *type*. *string* is assumed to represent a 32-bit numeric value in the C language numeric literal format. *type* must be an X-atom resource.

epoch::resource-to-type *resource* Function

Returns an X resource whose type is `atom` and whose value is the type of the *resource*.

epoch::resource-to-string *resource* &optional *base* Function

Convert the id of the *resource* to a numeric string. Optional *base* specifies the base for the conversion (may be 2..36 inclusive). Note that no special leading characters are added for non-decimal bases.

epoch::xid-of-screen &optional *screen* Function

Returns the X-window resource value for the *screen* (default: current-screen).

epoch::set-resource-type *resource type* Function

Sets the type field of *resource* to the value of *type*. *type* must be an X-atom resource. Returns *resource* if it was successfully modified, nil otherwise.

6.2 Other Stuff

epoch::rebind-key *keysym shiftmask string* Function

Rebinds a raw X key on the fly. Takes as arguments the *keysym* to rebind, the *shiftmask* for the rebinding, and a *string* to send when that key is depressed. This function does not affect keybindings for other X clients.

epoch::mod-to-shiftmask *index* Function

Input is an X modifier *index*, output a shiftmask that can be passed to `epoch::rebind-key`.

epoch::query-tree &optional *screen-or-xwin* Function

Returns a list of the form `(root parent . children)` where all the elements are X-window resources. `root` is the X root window, `parent` is the X parent window of *screen-or-xwin* (`nil` if *screen-or-xwin* is the X root window), and `children` is a list of the X child windows of *screen-or-xwin*.

epoch::set-bell *arg* Function

If *arg* is `nil`, Epoch will use an audible bell. If non-`nil`, then a visual bell will be used.

epoch::bell-volume Variable

Controls the X bell volume. If a number from -100 to 100, then the value is used as the bell volume, otherwise it is ignored and 50 is used. Note that on many systems, the hardware does not support different bell volumes, and so this may not be effective.

6.2.1 Client Messages

epoch::send-client-message *to* &optional *from data type format* Function

Sends a client message of type *type* to the destination *to* using *format* and *data*. The parameters have the following meaning:

to A screen or an X-resource of type window. See Chapter 6 [X11 Primitives], page 45 for more information.

from A screen or an X-resource of type window that serves as the source of the message. This defaults to the current screen.

data The message data may be `nil` if there is no data, or a string, or a list or vector of integers or X-resources. If it is simply a string, the first 20 characters are transmitted, and interpreted as 1, 2 or 4-byte data on the other end according to the format parameter. If there is no data, `0`s are sent.

type This is an X-atom (as opposed to an elisp atom) and can be created by `intern-atom`. Usually something which conveys the meaning of the message, e.g., `WM_CHANGE_STATE`.

format The data for a client message is always 20 bytes long. The format statement tells how to interpret this. The value must be one of the following:

`nil` The format defaults to a value appropriate for the data.

`8` for 20 1-byte data objects.

`16` for 10 2-byte objects.

`32` for 5 4-byte objects.

```
;; Fake an iconify message
(setq xroot (car (query-tree))) ; get the root window
;; IconicState is 3
;; Send to the root window, from the current screen,
;; with data 3 of type WM_CHANGE_STATE.
(send-client-message xroot nil 3
  (intern-atom "WM_CHANGE_STATE"))
```

6.2.2 X Properties

epoch::get-property *name* &optional *screen* Function

Returns the X-property *name* on the *screen*. *screen* can be an X-window resource. *name* can be an X-atom resource. The property value is returned in the most (in

Epoch's opinion) convenient form.

If the property is an array of items (e.g., the *count* return value is more than 1) then it is returned as a list. Strings are converted to Elisp strings, integers into Elisp integers (with a possible loss of precision, since Elisp integers are less than 32 bits), and other 32-bit types into X-resources. Strings are handled specially - if nulls are found in the returned byte array, they are assumed to represent string seperators, and that the property is an array of these null-seperated substrings.

Returns the property value or `nil` if there is a failure (bad screen, no property, bad type, etc.).

epoch::set-property *name value* &optional *screen* Function

Sets the X-property *name* to *value* on the X-window *screen*. *screen* can be an X-window resource. *name* can be an X-atom resource. *value* must be a string, an integer, an X-resource, or an vector or list of these types. If it is a vector or list, the types of all the sequence elements must be the same, and for X-resources the X-resource types must also be the same.

6.2.3 X Cursor

epoch::query-pointer &optional *screen* Function

Returns a list containing x-coordinate, y-coordinate (in pixels) and state of mouse. Uses current screen if no *screen* argument is given.

epoch::warp-pointer *x y* &optional *screen* Function

Warps cursor to (*x*,*y*) location (in pixels) on *screen*. Uses current screen if no *screen* argument is given.

epoch::query-mouse &optional *screen* Function

Returns a list containing x-coordinate, y-coordinate (in character position) and state of mouse. Uses current screen if no *screen* argument is given.

epoch::warp-mouse *x y* &optional *screen* Function

Warps cursor to (*x*,*y*) location (in character position) on *screen*. Uses current screen if no *screen* argument is given.

7. Miscellaneous

Several unrelated features of Epoch are described here.

Epoch contains some extensions of GNU Emacs that are not specifically related to Epoch.

symbol-buffer-value *symbol buffer* Function

Returns the value of *symbol* in *buffer*, without the expense of using `set-buffer`.

epoch::function-key-mapping Variable

If this variable is non-`nil`, then functions keys will be mapped into an extended form starting with ESC [, as is done in normal GNU-Emacs. If set to `nil`, then no mapping will be done, and the key will be ignored. Keys rebound (see Section 6.2 [Other Stuff], page 46, `epoch::rebind-key`) to a non-empty string will not be affected by this mapping.

7.1 Epoch Version

The built-in variable `epoch::version` will return the version number of an epoch editor. Thus, `(boundp 'epoch::version)` will be `t` on epoch and `nil` on any other version of GNU Emacs. You can use this to customize your elisp code.

7.2 Packages

Epoch comes with a set of standard elisp files, which provide functionality to the user. The functions and facilities are discussed here. The following files are included in this version of the Epoch distribution, in the `epoch-lisp-files` directory:

'`.emacs`' Things that should go in your `.emacs` file. Pretty straight-forward; read the code.

'`epoch.el`'

Interfaces to primitive, Epoch-specific lisp functions, standard keybindings, variables that you can set to change things, etc. This file also defines the default event handler code.

Every primitive function introduced into GNU Emacs to support Epoch is of the form `epoch::`*name*. This is like common lisp internal package symbols, and for each we

provide a form without the leading `epoch::`. These are the forms you should use; this way, you can add additional functionality to the primitives very easily. For example, the definition of `create-screen` determines what to take from the `epoch-mode-alist` before creating the screen with `epoch::x-create-screen`.

'`maintain-screen-titles.el`'
When loaded, each screen's title dynamically changes to reflect the name of the current buffer and the name of the machine you are running on. It rebinds some standard key bindings to get this effect. For any function `foo` that changes a buffer, produce a `my-foo` with the same effect, but renaming the screen along the way. Also this file defines `C-za` to rename current screen, which is useful if things get out of sync.

Not currently loaded.

'`mouse.el`'
Mouse event handling for epoch.

'`motion.el`'
Handles mouse motion events. Provides dragging and pasting operations.

'`property.el`'
Provides the basic property handlers.

'`cute-wins.el`'
Create screens for '`mh-rmail`' and '`calendar`' programs to run in.

Not currently loaded.

'`server.el`'
Modified version of the standard '`server.el`' program, pops a new window when you use Epoch as a server and call it through `emacsclient`. Works identically to old '`server.el`' code, with exception of this new feature, *which is conditional on Epoch running, so you can use it with all versions of GNU Emacs.*

'`message.el`'
Provides a client message handler. Provides a handler for the WM_DELETE_WINDOW client message from the window manager.

7.2.1 Mouse Dragging

We have implemented drag, scrolling drag, cut and paste with the mouse. The files '`mouse.el`' and '`motion.el`' must be loaded to use this feature.

The functionality of buttons with this feature loaded is shown in following table.

Button	Action
left-down	Clear drag region, set point
left-down and drag	Drag out a region. Highlight (buffer-local), place in cut-buffer and kill-ring. Set point and mark around highlighted region.
middle-down	Paste cut-buffer at the mouse location. (Use `yank` to paste at the current point.)
right-down	Extend highlighted region from a left-down and drag.
right-down and drag	Extend highlighted region initially, then adjust by drag.

Scrolling Drag means that when the mouse moves out the window during drag the scroll continues until the mouse-button is raised. The speed of the drag can be controlled by setting the following variables:

horizontal-drag-inc 5 Variable

The number of characters the buffer is scrolled horizontally in a single scroll.

vertical-drag-inc 2 Variable

The number of lines the buffer is scrolled vertically in a single scroll.

Dragged regions are buffer-local, so you can have drags in multiple buffers simultaneously. Because point and mark are set around drags, you can use a combination of mouse and Epoch keystrokes to do editing.

7.2.2 Screen Naming

Set the variable `include-system-name` to `t` if you would like screen names to include the name of the system they are running on, for example the name of the screen with buffer `foo` selected on machine `bar` would appear `foo @ bar`.

7.2.3 Autoraise

Setting the variable `auto-raise-screen` to `t` causes screens to raise when they are selected with `x-select-screen`.

7.2.4 Multiple Screen Updates

The '`epoch.el`' file sets things up so all screens are updated automatically. You will have to change this if you would like a different convention. For example, you could use the `epoch-mode-alist` to set things up so that only screens created for buffers with mode `shell-mode` would be updated. See Section 3.5 [Screen Updating], page 22 for more information.

7.3 Screen Pools

The package in '`scr-pool.el`' provides a set of functions for creating pools of screens. This is a set of screens of a fixed size that are used in a package. Screens pools are useful when a package wants to use multiple screens, but wants a limit on the number of screens in use to prevent excessive clutter or resource consumption.

7.3.1 Screen Pool Basics

A screen pool consists of

Size The maximum number of screens allowed in the pool. When this many screens are in the pool, and another is requested, one of the screens already in the pool is recycled and returned instead of a new screen.

Create-Function
This is a function that is used to create a new screen when needed, either because there are less than *size* screens in the pool, or one of the screens in the pool is dead. This function is optional, and if missing `create-screen` is used.

Cleanup-Function
This function is called on a screen just before it is recycled. If missing, no function is called. This is useful, for example, when buffers displayed in a screen should be killed when the screen is recycled.

The screen pool attempts to keep track of the least recently used screen and will recycle that screen first. A screen is marked as used whenever it is returned as a requested screen. In addition, there are functions to explicitly mark screens as either most or least recently used. These function should not be called on screens not in the pool, since this will cause the screen to be added to the pool.

7.3.2 Screen Pool Functions

These are the functions provided by the Screen Pool package.

pool:create *size* &optional *create-function cleanup-function* Function

Creates and returns a screen pool with no screens in it.

pool:delete *pool* Function

Deletes all the screens in the *pool* using `delete-screen`. The cleanup-function, if any, is called on each screen first.

pool:get-screen *pool* Function

Returns a screen from the *pool*. If there are fewer screens than the maximum size, a new screen is created, otherwise an existing screen is recycled. If a screen is recycled, the cleanup-function (if any) is called on the screen.

pool:get-screen-with-buffer *pool buffer* Function

Similar to `pool:get-screen`, except that if one of the screens in the pool is already displaying *buffer*, it is returned instead of creating a new screen or recycling another screen, and the cleanup-function is not called.

pool:get-shrink-wrapped-screen *pool buffer limits* Function

Similar to `pool:get-screen`. If *buffer* is displayed on any screen in the *pool*, then that screen is used, otherwise a screen selected as in `pool:get-screen` is used. This screen is selected, all but one window is deleted, and that window is set to display *buffer*. The screen is then shrunk to fit the buffer, in height and width, up to *limits*. *limits* should be a list of 4 numbers, of the form `(min-width, max-width, min-height, max-height)`.

pool:mark-screen *pool screen* Function

Mark *screen* as being the most recently used *screen* in the *pool*. This means it will be the last to be recycled.

pool:unmark-screen *pool screen* Function

Mark *screen* as being the least recently used *screen* in the *pool*. This means it will be the first screen to be recycled.

7.4 Colors

Colors can be specified as either a string, which is used for lookup in the X color database, or a vector with three components, the red, green, and blue components. Vector values are such that `65535 (==2^16-1)` is the maximum for the display device, and 0 is off. In general, Epoch uses the vector as a color descriptor, and functions that return colors return a vector.

x-number-of-colors Function

Returns the number of color cells on the display. This function is used to distinguish between monochrome and color systems. A result of `2` indicates monochrome, a larger number indicates a grey-scale or color display.

get-color *name* Function

Given a color name, this function returns a three element vector containing the red, green, and blue components of the color. It returns `nil` if the color lookup fails.

7.5 Change Hooks

Two variables were added to Epoch that hold the names of functions to call before and after, respectively, any changes are made to a buffer. These variables are not buffer-local, so if you wish to have your change function called only for specific buffers, first make the variables buffer-local, then assign the name of the function to be called.

before-change-function Variable

Function to call before each text change. Two arguments are passed to the function: the position of the change and the position of end of the region deleted.

While executing the `before-change-function`, changes to buffers do not cause calls to any `before-change-function` or `after-change-function`.

after-change-function Variable

Function to call after each text change. Three arguments are passed to the function: the position of the change, the position of the end of the region inserted, and the length of the deletion.

While executing the `after-change-function`, changes to buffers do not cause calls to

any `before-change-function` or `after-change-function`.

```
(defun after-change (pos inspos dellen)
  "Called after each change"
  (condition-case err
      (progn
;      (garbage-collect)
       (cond ((= dellen 0)
              (message "insertion: (%s %s)" pos inspos))
             ((= dellen (- inspos pos))
              (message "replacement: (%s %s)" pos inspos))
             ((= pos inspos)
              (message "deletion: %s of length %s" pos dellen))
             (t
              (message "other change at %s: ins %s, del %s"
                       pos inspos dellen))
             )
       )
    (error
     (setq after-change-function nil)
     (message "*Change error: %s" (prin1-to-string err)))
    )
  )

;; first evaluate the above function
;; then eval the next two lines to activate the function

(make-local-variable 'after-change-function)
(setq after-change-function 'after-change)
```

7.6 Icons

You can set the names of icons for screens (accessible through the property `WM_ICON_NAME` using the global `.Xdefaults`, the screen property list or the function `epoch::icon-name`.

epoch::icon-name &optional *value screen* Function

If value is `nil`, return current icon name. Otherwise set icon name to value. If screen is `nil`, use current screen.

7.7 dbx

dbx Function

Calls a C function named DEBUG which does nothing. This serves as an entry into a debugger.

Index

$

$HOME/.Xdefaults 5
$HOME/.Xdefaults-<hostname> 5

*

create-screen-alist-hook 14

A

add-read-only-button 30
after-change-function 54

B

background color 8
before-change-function 54
border color 16
bug reports 1
button attributes 27
button creation 25
button foreground and background colors 27
button object 25
button plotting 28
button-text 27
buttons 25
buttons on monochrome screen 27

C

color vector or string 11
command line options 8
copy-mouse-map 41
create screen 13, 14
create-mouse-map 41
create-screen 14
create-screen properties alist 15
current screen 13
cursor background color 16
cursor colors in button 28
cursor foreground color 16
cursor glyph 16

D

dbx 55
default sets 5
defaults in command line options 5
define-mouse 41
deleting buttons 27
display-event-status 37

E

edit screens 13
epoch-mode-alist 15
epoch-mode-alist properties alist 15
epoch:: 11
epoch::add-button 26, 30
epoch::background 22
epoch::bell-volume 46
epoch::button-at 26
epoch::button-attribute 28
epoch::button-buffer 26
epoch::button-data 26
epoch::button-end 26
epoch::button-list 26
epoch::button-read-only 31
epoch::button-start 26
epoch::buttonp 25
epoch::buttons-modify-buffer 27
epoch::change-screen-size 20
epoch::clear-buttons 27
epoch::coords-to-point 42
epoch::create-screen 14
epoch::create-screen properties alist 15
epoch::current-screen 18
epoch::cursor-color 22
epoch::cursor-glyph 22
epoch::delete-button 27
epoch::delete-button-at 27
epoch::delete-screen 18
epoch::event 36
epoch::event-handler 35
epoch::event-handler-abort 36
epoch::first-window 19
epoch::flash-screen 22
epoch::font 21

epoch::foreground ... 21
epoch::function-key-mapping ... 49
epoch::get-buffer-window ... 18
epoch::get-event ... 34
epoch::get-property ... 47
epoch::get-screen ... 17
epoch::get-screen-id ... 17
epoch::global-update ... 23
epoch::icon-name ... 55
epoch::iconify-screen ... 20
epoch::intern-atom ... 45
epoch::lazy-events ... 34
epoch::local-buttons-p ... 29
epoch::lower-screen ... 20
epoch::make-button ... 25
epoch::map-screen ... 20
epoch::mapraised-screen ... 20
epoch::minibuf-screen ... 18
epoch::mod-to-shiftmask ... 46
epoch::motion-hints-p ... 43
epoch::mouse-events ... 34
epoch::move-button ... 26
epoch::move-screen ... 21
epoch::next-screen ... 17
epoch::plane-size ... 21
epoch::prev-screen ... 17
epoch::query-mouse ... 48
epoch::query-pointer ... 48
epoch::query-tree ... 46
epoch::raise-screen ... 19
epoch::read-only-region-p ... 30
epoch::rebind-key ... 46
epoch::redisplay-screen ... 23
epoch::resource-to-string ... 46
epoch::resource-to-type ... 45
epoch::resourcep ... 45
epoch::screen-height ... 20
epoch::screen-information ... 21
epoch::screen-list ... 17
epoch::screen-mapped-p ... 20
epoch::screen-of-window ... 19
epoch::screen-p ... 17
epoch::screen-properties ... 15
epoch::screen-properties alist ... 15
epoch::screen-width ... 20
epoch::screenp ... 17
epoch::screens-of-buffer ... 19
epoch::select-screen ... 17
epoch::selected-window ... 19
epoch::send-client-message ... 47
epoch::set-attribute-global ... 29
epoch::set-attribute-local ... 29
epoch::set-bell ... 46
epoch::set-button-attribute ... 28
epoch::set-button-data ... 26
epoch::set-button-read-only ... 31
epoch::set-local-buttons ... 29
epoch::set-motion-hints ... 43
epoch::set-property ... 48
epoch::set-resource-type ... 46
epoch::set-screen-modified ... 23
epoch::set-update ... 23
epoch::string-to-resource ... 45
epoch::unintern-atom ... 45
epoch::unmap-screen ... 20
epoch::update-p ... 23
epoch::warp-mouse ... 48
epoch::warp-pointer ... 48
epoch::xid-of-screen ... 46
external and internal borderwidth of screen ... 21
external border width ... 16

F

font ... 8, 16
foreground color ... 8

G

generic window manager ... 11
get-color ... 54
global button attributes ... 28
global-set-mouse ... 42
GNU Emacs compatibility ... 1
GWM ... 11

H

hooks ... 14

horizontal-drag-inc 51
hypertext 25

I

ICCCM compatability for deletion 18
ICCCM compatibilty for iconify 20
icon name 16
IconicState 16
ignore-event 36
ignore-message 39
ignore-property 38
install-event 36
install-message 39
install-property 38
internal border width 16

K

key bindings 9
kill-local-mouse-map 41

L

local button attributes 28
local-set-mouse 42

M

mailing list support 1
make-id 17
make-screen 17
mapped screen 13
minibuffer background 6
minibuffer border color 6
minibuffer border width 6
minibuffer class 7
minibuffer cursor color 6
minibuffer cursor foreground 6
minibuffer cursor glyph 7
minibuffer font 7
minibuffer foreground 7
minibuffer geometry 7
minibuffer internal border width 7
minibuffer name 7
minibuffer resource name 7
minibuffer reverse 7
minibuffer screens 13
minibuffer title 7
minor screens 21
motion events 7, 8, 16
mouse::convert-xy-screen 42
mouse::set-point 42

N

NormalState 16

O

on-map-do 43
on-move-do 43
on-resize-do 43
on-unmap-do 43
overlapping buttons 28

P

pool:create 53
pool:delete 53
pool:get-screen 53
pool:get-screen-with-buffer 53
pool:get-shrink-wrapped-screen 53
pool:mark-screen 53
pool:unmark-screen 53
pop-event 36
pop-message 39
pop-property 38
push-event 36
push-message 39
push-property 38

R

read-only buttons 30
release-attribute 30
release-attributes 30
remove-event 36
remove-message 39
remove-property 38
reserve-attribute 29
reserve-attributes 29
resource class 8
resource manager 8

resource name . . . 8
`resume-event` . . . 36
`resume-message` . . . 39
`resume-property` . . . 38
reverse . . . 16
reverse colors . . . 8

S

screen background . . . 6, 16
screen border color . . . 6
screen border width . . . 6
screen class . . . 7
screen class name . . . 16
screen cursor background . . . 6
screen cursor color . . . 6
screen cursor foreground . . . 6
screen cursor glyph . . . 7
screen default-sets . . . 14
screen font . . . 7
screen foreground . . . 7, 15
screen geometry . . . 7, 8, 16
screen icon name . . . 8
screen ID or screen object . . . 10
screen initial state . . . 16
screen internal border width . . . 7
screen mapped state . . . 21
screen name . . . 7
screen object . . . 13
screen properties or attributes . . . 14
screen resource name . . . 7, 16
screen reverse . . . 7
screen title . . . 7, 16
screen updating . . . 16, 22
selected screen . . . 13
`symbol-buffer-value` . . . 49

T

text cursor color . . . 16

U

unmapped screen . . . 13
unmapped screens list . . . 17
`use-local-mouse-map` . . . 41

V

`vertical-drag-inc` . . . 51

W

width and height of screen . . . 21
window configuration . . . 13
window title . . . 8
windowing system controls . . . 19

X

X and Y location of screen . . . 21
X class defaults . . . 6
X parent resource id . . . 16
X program or window class name . . . 5
X program or window resource name . . . 5
X resources database . . . 5
X window display . . . 8
X Window display . . . 7
X Window program defaults . . . 5
X windows and GNU Emacs windows . . . 13
X Windows required . . . 1
`x-number-of-colors` . . . 54
`XENVIRONMENT` . . . 5
`xrdb` . . . 5

Z

zero button attributes . . . 28

Short Contents

1 Introduction 1
2 General Information 5
3 Screens 13
4 Buttons 25
5 Events 33
6 X11 Primitives 45
7 Miscellaneous 49
Index 57

Table of Contents

1 Introduction 1

1.1 Support and Mailing List 1
1.2 Getting Epoch 2
1.3 Acknowledgements 2

2 General Information 5

2.1 Getting Started 5
2.1.1 X Window Program Defaults 5
2.1.2 Command Line Options 8
2.2 New Features 9
2.3 Key Bindings 9
2.4 Conventions 10
2.5 GWM 11

3 Screens 13

3.1 Screen Properties 14
3.2 Controlling Screens 16
3.3 Screens and Windows 18
3.4 Manipulating Screens 19
3.5 Screen Updating 22

4 Buttons 25

4.1 Button Primitives 25
4.2 Deleting Buttons 27
4.3 Button Attributes 27
4.4 Attribute Allocation 29
4.5 Read-only Buttons 30

5 Events 33

5.1 Event Basics 33
5.2 Event Handling 34
5.3 Property Events 37
5.4 Client Message Events 38
5.5 Mouse Events 39
5.6 Motion Events 42
5.7 On Event Handling 43

6 X11 Primitives 45

6.1 X Resources 45
6.2 Other Stuff 46
6.2.1 Client Messages 47
6.2.2 X Properties 47
6.2.3 X Cursor 48

7 Miscellaneous 49

7.1 Epoch Version 49
7.2 Packages 49
7.2.1 Mouse Dragging 50
7.2.2 Screen Naming 51
7.2.3 Autoraise 51
7.2.4 Multiple Screen Updates 52
7.3 Screen Pools 52
7.3.1 Screen Pool Basics 52
7.3.2 Screen Pool Functions 53
7.4 Colors 54
7.5 Change Hooks 54
7.6 Icons 55
7.7 dbx 55

Index 57