CLUX: XLIB for PCLU

CONTENTS

INTRODUCTION

Clux is a new and actually usable new interface from Clu to X. It's based on Xlib, the C interface to X, but hides some of Xlib's complexity (often by neglecting to implement it, yay). To learn Clux, have a look at the specifications, below. This'll be a little overwhelming; it may be easier to look through the code which follows the specifications and refer back to the relevant sections of the specs. The code is for xshostakovich, a "real-world" (ha!) application written with Clux which demonstrates a large number of Clux concepts in action. To run xshostakovich, add PCLU on an Athena DECstation or Sun workstation and type xshostakovich &.

When you compile with Clux, you'll need to know about three files: clux.equ, clux.lib, and libclux.a. Clux.equ contains many equates used by Clux; clux.lib contains specifications for all the Clux clusters and operations; and libclux.a contains the actual Clux code. A session with PCLU to build xshostakovich looks something like:

```
PCLU clux.equ –create the.lib clux.lib xshostakovich.clu    % Create our specifications library,
                                                            % with both Clux's and our specs
PCLU clux.equ the.lib –compile xshostakovich.clu            % Compile (remember Clux's equates)
PCLU –link xshostakovich xshostakovich.o libclux.a          % Link with the Clux library
```

display = **data type is** open, open_named, close, equal, get_width, get_width_millimeters, get_height, get_height_millimeters, get_depth, get_resource_database, get_root_window, get_focus_window, get_black, get_white, sync, query_pointer, next_event, mask_event, check_mask_event, check_typed_event, beep

A display is pretty much the specification of a monitor. You must have a display before you can do anything else related to X. All the routines which take a display require that it has not been closed.

open = **proc**() **returns** (display) **signals** (*cannot*)
open_named = **proc**(*name*: string) **returns** (display) **signals** (*cannot*)
effects Connects your program to an X server. Open chooses the default X server: the one specified by the DISPLAY environment variable. Open_named chooses the named X display. Signals *cannot* if the display cannot be opened for any reason.

close = **proc**(*d*: display)
effects Closes the display *d*. This removes all windows you have created on that display, and makes the display invalid, as well as all objects created from that display. Using an invalid object – display or whatever – will make your program die.

equal = **proc**(*d1*, *d2*: display) **returns** (bool)
effects Returns true if and only if *d1* and *d2* describe the same display object.

get_width = **proc**(*d*: display) **returns** (int)
get_width_millimeters = **proc**(*d*: display) **returns** (int)
get_height = **proc**(*d*: display) **returns** (int)
get_height_millimeters = **proc**(*d*: display) **returns** (int)
effects Return the dimensions of the given display's monitor. The 'millimeters' functions return the width or height in millimeters; the other functions return the width or height in pixels.

get_depth = **proc**(*d*: display) **returns** (int)
effects Returns the depth of the given display's monitor. The monitor can display 2^{depth} different colors.

get_resource_database = **proc**(*d*: display) **returns** (resource_database) **signals** (*no_database*)
effects Returns a copy of the resource database stored on the given display. Signals *no_database* if no database exists on that display. A display's resource database usually contains the resources that the user has set from her ~/.Xdefaults or ~/.Xresources file.

get_root_window = **proc**(*d*: display) **returns** (window)
effects Returns the root window of the given display.

get_focus_window = **proc**(*d*: display) **returns** (window)
effects Returns the window of the given display which has the keyboard focus: that is, the window to which keyboard events are being reported. If *d*.focus_window = *d*.root_window, the keyboard focus is pointer-relative: that is, the window the pointer is in gets the keyboard focus.

get_black = **proc**(*d*: display) **returns** (color)
effects Returns a color which appears black on the given display.

get_white = **proc**(*d*: display) **returns** (color)
effects Returns a color which appears white on the given display.

sync = **proc**(*d*: display)
  effects When something is drawn onto a display, the display saves the request to draw until 'later,' for efficiency reasons. If you don't want the display to wait, you can call display$sync to force the display to process all the drawing request it's saved up.

query_pointer = **proc**(*d*: display) **returns** (int, int)
  effects Returns the position the mouse pointer is at on the given display as x and y coordinates (in that order). The position is relative to the upper-left corner of *d*'s root window.

next_event = **proc**(*d*: display) **returns** (event)
mask_event = **proc**(*d*: display, *mask*: event_mask) **returns** (event)
  effects Returns the next matching event on the given display. If no matching event is ready, waits for one to be received and returns it. Next_event matches any event whatsoever; mask_event matches only events selected by the given event_mask (page 20).

check_mask_event = **proc**(*d*: display, *mask*: event_mask) **returns** (event) **signals** (*none*)
check_typed_event = **proc**(*d*: display, *match*: event) **returns** (event) **signals** (*none*)
  effects Returns the next matching event on the given display. Signals *none* immediately if no matching event is ready. Check_mask_event matches events selected by the given event_mask (page 20); check_typed_event matches events with the same species as *match*.

beep = **proc**(*d*: display, *howloud*: int)
  effects Makes *d* beep. *Howloud* should be a percentage: how loud the beep should be. It is constrained to lie between -100 (quiet) and 100 (loud). 0 is the best value for most purposes.

end display

window = **data type is** make_none, is_none, make_pointer_root, is_pointer_root, is_normal, create, create_toplevel, destroy, equal, get_display, get_position, get_position_x, get_position_y, move, get_dimensions, get_width, get_height, resize, get_depth, get_border_width, set_border_width, set_cursor, unset_cursor, map, unmap, raise, lower, map_raised, select_input, query_pointer, warp_pointer, grab_keyboard, ungrab_keyboard, grab_focus, grab_pointer, ungrab_pointer, mask_event, check_mask_event, check_typed_event, set_background_color, set_background_pixmap, clear, clear_area, set_border_color, set_border_pixmap, w2p, draw_point, draw_line, draw_rectangle, fill_rectangle, draw_lines, fill_polygon, draw_arc, fill_arc, draw_string, draw_image_string, copy_plane, copy_area

A window is an area of the screen which you can draw into. A window can be *top-level*: top-level windows are what people normally think of as "windows"; they are given title bars, etc., by the window manager, and there is usually one top-level window per application. However, windows do not have to be top-level. Windows are created hierarchically: each window, except the root window (see display$get_root_window, page 3), has a parent. Top-level windows are children of the root window; but you can create new windows that are children of a top-level window, and so on. Many applications use these child windows to implement buttons, scrollbars, and the like.

Aside from actual windows, there are two special kinds of window object which are useful for some operations: the *none* object and the *pointer_root* object. All calls with a window parameter require that the window is an actual window object, not *none* or *pointer_root*, unless otherwise noted. All calls also require that any window parameters have not been destroyed; see destroy, page 5.

make_none = **proc**(*d*: display) **returns** (window)
effects Creates and returns a *none* window for the given display.

is_none = **proc**(*w*: window) **returns** (bool)
requires *w* may be any window object.
effects Returns true if and only if *w* is a *none* window.

make_pointer_root = **proc**(*d*: display) **returns** (window)
effects Creates and returns a *pointer_root* window for the given display.

is_pointer_root = **proc**(*w*: window) **returns** (bool)
requires *w* may be any window object.
effects Returns true if and only if *w* is a *pointer_root* window.

is_normal = **proc**(*w*: window) **returns** (bool)
requires *w* may be any window object.
effects Returns true if and only if *w* is an actual window object, not *none* or *pointer_root*.

create = **proc**(*parent*: window, *x*, *y*, *w*, *h*: int) **returns** (window)
effects Creates a window with *parent* as its parent window. The window's upper-left corner is at (*x*, *y*) relative to *parent*'s upper-left corner; it is *w* pixels wide and *h* pixels high.

create_toplevel = **proc**(*d*: display, *w*, *h*: int, *windowname*, *resourcename*, *resourceclass*: string, *input_mask*: event_mask) **returns** (window) **signals** (*cannot*)
effects Creates a top-level window on the given display. The window is *w* pixels wide and *h* pixels high, unless the user resizes it. The window's title and its icon's name are both *windowname*. Resources can be specified for it with the resource name *resourcename* and resource class *resourceclass*. Initially, the window accepts input events which are selected by the mask *input_mask* (see select_input, page 6).

destroy = **proc**(*w*: window)
modifies *w*.
effects Destroys the given window. *W* is removed from the screen and all memory associated with it is freed; also, any window objects referring to it become invalid, and cannot be used.

equal = **proc**(*w1*, *w2*: window) **returns** (bool)
requires *w1* and *w2* may be any window objects.
effects Returns true if and only if *w1* and *w2* refer to the same window.

get_display = **proc**(*w*: window) **returns** (display)
effects Returns the display *w* was created on.

get_position = **proc**(*w*: window) **returns** (int, int)
get_position_x = **proc**(*w*: window) **returns** (int)
get_position_y = **proc**(*w*: window) **returns** (int)
effects Return the position of the window *w* relative to its parent's upper-left corner. Get_position returns both the x and y coordinates, in that order.

move = **proc**(*w*: window, *x*, *y*: int)
modifies *w*.
effects Moves *w* to a position of (*x*, *y*) relative to its parent's upper-left corner.

get_dimensions = **proc**(*w*: window) **returns** (int, int)
get_width = **proc**(*w*: window) **returns** (int)
get_height = **proc**(*w*: window) **returns** (int)
effects Return the dimensions of the window *w*. Get_dimensions returns both width and height, in that order.

resize = **proc**(*w*: window, *width*, *height*: int)
requires *width* > 0, *height* > 0.
modifies *w*.
effects Resizes *w* to *width* pixels wide and *height* pixels high.

get_depth = **proc**(*w*: window) **returns** (int)
effects Returns the depth of the window *w*. A total of 2^{depth} different colors can be used on *w*.

get_border_width = **proc**(*w*: window) **returns** (int)
effects Returns the width of *w*'s border.

set_border_width = **proc**(*w*: window, *bw*: int)
requires *bw* ≥ 0.
modifies *w*.
effects Sets the width of *w*'s border to *bw*.

set_cursor = **proc**(*w*: window, *c*: cursor) **signals** (*display_conflict*)
effects Sets the cursor displayed when the pointer is in *w* to *c*. Signals *display_conflict* if *w* and *c* were not created on the same display. See cursor, page 15.

unset_cursor = **proc**(*w*: window)
effects Removes any cursor definition from *w*. The cursor displayed when the pointer is in *w* will be the same as the cursor displayed when the pointer is in *w*'s parent.

map = **proc**(*w*: window)
unmap = **proc**(*w*: window)
effects Maps or unmaps the window *w*. A window can be visible on the screen only if it is mapped.

raise = **proc**(*w*: window)
lower = **proc**(*w*: window)
effects Raises or lowers the window *w*. Raising *w* puts it "above" all the other windows which have the same parent; lowering *w* puts it "below" them.

map_raised = **proc**(*w*: window)
effects Maps the window *w* on top of all the other windows which have the same parent. Useful for top-level windows. See map and raise, above.

select_input = **proc**(*w*: window, *input_mask*: event_mask)
effects Makes *w* only able to report events which are selected by *input_mask*. See event_mask, page 20.

query_pointer = **proc**(*w*: window) **returns** (int, int) **signals** (*not_in_window*)
effects Returns the position of the pointer relative to the upper-left corner of *w* as x and y coordinates (in that order). Signals *not_in_window* if the pointer is not in *w*.

warp_pointer = **proc**(*w*: window, *tox*, *toy*: int)
effects Instantaneously moves the pointer to the position (*tox*, *toy*), relative to the upper-left corner of *w*.

grab_keyboard = **proc**(*w*: window) **signals** (*already_grabbed*, *cannot*)
effects Grabs the keyboard so that all keyboard events are reported to *w*, no matter what the keyboard focus is (see display$get_focus_window, page 3). If another program has already grabbed the keyboard, signals *already_grabbed*. If the grab fails for some other reason, signals *cannot*.

ungrab_keyboard = **proc**(*w*: window)
effects Releases a the keyboard from a grab by the window *w*. After ungrab_keyboard is called, keyboard events are reported normally.

focus_revert_to = **oneof**[parent, pointer_root, none: null]
grab_focus = **proc**(*w*: window, *revert_to*: focus_revert_to)
requires *w* may be *none* or *pointer_root*. If *w* is an actual window, it must be viewable (it and its parents must all be mapped).
effects Sets the keyboard focus window to *w*, so that all further keyboard events will be reported to *w* or one of its children. If *w* is *none*, no window gets the keyboard focus, so all keyboard events are discarded. If *w* is *pointer_root*, the focus window is set to the root window of the screen the pointer is on. If *w* is an actual window, *revert_to* specifies what happens to the keyboard focus if *w* becomes unviewable: parent means the keyboard focus reverts to the closest viewable ancestor of *w*; none or pointer_root act like passing arguments of *none* or *pointer_root* to grab_focus.

grab_pointer = **proc**(*w*, *confine_to*: window, *c*: cursor, *mask*: event_mask) **signals** (*display_conflict*, *already_grabbed*, *cannot*)
requires *confine_to* and *c* may be *none*.
effects Grabs the pointer so that all pointer motion and button events are reported to *w*, no matter where the pointer is. The pointer is prevented from moving outside *confine_to* if *confine_to* is not *none*. The cursor is changed to *c* if *c* is not *none*. Only the events specified by *mask* are reported while the grab is active. Signals *display_conflict* if *w*, *confine_to*, and *c* were not all created with the same display; *already_grabbed* if the pointer has already been grabbed by another program; and *cannot* if the grab fails for some other reason.

ungrab_pointer = **proc**(*w*: window)
effects Releases a the pointer from a grab by the window *w*. After ungrab_pointer is called, pointer events are reported normally.

mask_event = **proc**(*w*: window, *mask*: event_mask) **returns** (event)
effects Returns the next matching event relevant to the given window. If no matching event is ready, waits for one to be received and returns it. Events match only if they have been selected by the given event_mask (page 20).

check_mask_event = **proc**(*w*: window, *mask*: event_mask) **returns** (event) **signals** (*none*)
check_typed_event = **proc**(*w*: window, *match*: event) **returns** (event) **signals** (*none*)
effects Return the next matching event relevant to the given window. Signal *none* if no matching event is ready. Check_mask_event matches events selected by the given event_mask (page 20); check_typed_event matches events with the same species as *match*.

set_background_color = **proc**(*w*: window, *c*: color) **signals** (*display_conflict*)
modifies *w*.
effects Sets the background color of *w* to *c*. Signals *display_conflict* if *w* and *c* were created on different displays.

set_background_pixmap = **proc**(*w*: window, *p*: pixmap) **signals** (*display_conflict*)
requires *p* may *none* or *parent_relative*.
modifies *w*.
effects Sets the background pixmap of the window *w* to *p*. This will take precedence over *w*'s background color. Signals *display_conflict* if *w* and *p* were created on different displays. A *none* pixmap argument means the window's background will be invisible (unless the background color has previously been set); a *parent_relative* argument means the same background pixmap as *w*'s parent will be used, but the origin of the tiling will be altered so that the join between *w*'s parent and *w* will be seamless.

clear = **proc**(*w*: window)
clear_area = **proc**(*w*: window, *x*, *y*, *width*, *height*: int)
requires *width* > 0, *height* > 0.
modifies *w*.
effects Erases *w*, or a rectangular portion of *w* specified by the upper-left corner (*x*, *y*) and the dimensions *width* and *height*. The areas erased are filled with the window's background; see set_background and set_background_pixmap, above.

set_border_color = **proc**(*w*: window, *c*: color) **signals** (*display_conflict*)
set_border_pixmap = **proc**(*w*: window, *p*: pixmap) **signals** (*display_conflict*)
requires *p* may be *none*.
modifies *w*.
effects Sets the border of the window *w* to either a solid color or a pixmap. Signals *display_conflict* if *w* and *c*, or *w* and *p*, were created on different displays.

w2p = **proc**(*w*: window) **returns** (pixmap)
effects Returns a pixmap object which refers to the image contained in *w*. This is useful for commands like copy_area (page 9) and pixmap$load_bitmap (page 11), which require a pixmap argument.

draw_point = **proc**(*w*: window, *gc*: gcontext, *x*, *y*: int) **signals** (*display_conflict*)
modifies *w*.
effects Draws a point in *w* with *gc* at the location (*x*, *y*). Signals *display_conflict* if *w* and *gc* were not created on the same display.

draw_line = **proc**(*w*: window, *gc*: gcontext, *x1*, *y1*, *x2*, *y2*: int) **signals** (*display_conflict*)
modifies *w*.
effects Draws a straight line in *w* with *gc* from (*x1*, *y1*) to (*x2*, *y2*). Signals *display_conflict* if *w* and *gc* were not created on the same display.

draw_rectangle = **proc**(*w*: window, *gc*: gcontext, *x*, *y*, *width*, *height*: int) **signals** (*display_conflict*)
requires *width* > 0, *height* > 0.
modifies *w*.
effects Draws a rectangle in *w* with *gc*. The rectangle has upper-left corner at (*x*, *y*) and dimensions *width* and *height*. Signals *display_conflict* if *w* and *gc* were not created on the same display.

fill_rectangle = **proc**(*w*: window, *gc*: gcontext, *x*, *y*, *width*, *height*: int) **signals** (*display_conflict*)
requires *width* > 0, *height* > 0.
modifies *w*.
effects Fills a rectangle in *w* with *gc*. The rectangle has upper-left corner at (*x*, *y*) and dimensions *width* and *height*. Signals *display_conflict* if *w* and *gc* were not created on the same display.

point = **struct**[x, y: int]
draw_lines = **proc**(*w*: window, *gc*: gcontext, *points*: array[point]) **signals** (*display_conflict*)
effects Draws a segmented line in *w* with *gc*. The line starts at the first point in *points*, moves to the second, etc., etc., and ends at the last. If the last point is the same as the first, the line is closed: the last segment and the first segment are joined just like any other pair of segments. Signals *display_conflict* if *w* and *gc* were not created on the same display.

fill_polygon = **proc**(*w*: window, *gc*: gcontext, *points*: array[point]) **signals** (*display_conflict*)
modifies *w*.
effects Fills a polygon in *w* with *gc*. The polygon's corners are the elements of *points* in order. Signals *display_conflict* if *w* and *gc* were not created on the same display.

draw_arc = **proc**(*w*: window, *gc*: gcontext, *x*, *y*, *width*, *height*: int, *angle1*, *angle2*: real)
signals (*display_conflict*)
requires *width* > 0, *height* > 0.
modifies *w*.
effects Draws an arc in *w* with *gc*. The arc is part of the largest ellipse which would fit in the rectangle with upper-left corner (*x*, *y*) and dimensions *width* and *height*. *Angle1* and *angle2* are in degrees. The arc is drawn starting from the angle position *angle1* (where 0 degrees is 3 o'clock – the normal Cartesian coordinate system), and subtends an angle of *angle2*, measured counterclockwise. Signals *display_conflict* if *w* and *gc* were not created on the same display.

fill_arc = **proc**(*w*: window, *gc*: gcontext, *x*, *y*, *width*, *height*: int, *angle1*, *angle2*: real)
signals (*display_conflict*)
requires *width* and *height* are positive.
modifies *w*.
effects Fills an arc in *w* with *gc*. The arc is part of the largest ellipse which would fit in the rectangle with upper-left corner (*x*, *y*) and dimensions *width* and *height*. *Angle1* and *angle2* are in degrees. The arc is drawn starting from the angle position *angle1* (where 0 degrees is 3 o'clock – the normal Cartesian coordinate system), and subtends an angle of *angle2*, measured counterclockwise. Signals *display_conflict* if *w* and *gc* were not created on the same display.

draw_string = **proc**(*w*: window, *gc*: gcontext, *x*, *y*: int, *s*: string) **signals** (*display_conflict*)
draw_image_string = **proc**(*w*: window, *gc*: gcontext, *x*, *y*: int, *s*: string) **signals** (*display_conflict*)
modifies *w*.
effects Draws a string in *w* with *gc*. The string's baseline begins at (*x*, *y*). Draw_string merely draws the foreground portions of each character in the string; draw_image_string erases the background portions of all the characters as well. Signals *display_conflict* if *w* and *gc* were not created on the same display.

copy_plane = **proc**(*w*: window, *p*: pixmap, *gc*: gcontext, *srcx*, *srcy*, *width*, *height*, *destx*, *desty*, *plane*: int)
signals (*display_conflict*)
requires *plane* = 2^x where $0 \le x < p$.depth.
modifies *w*.
effects Copies one plane of *p* onto *w* using *gc*: *gc*'s foreground color is used for bits of the plane which are set, its background color for bits which are cleared. Only the portion of *p* which begins at (*srcx*, *srcy*) and has dimensions *width* and *height* is copied; that portion is placed on *w* with its upper-left corner at (*destx*, *desty*). Signals *display_conflict* if *w*, *p*, and *gc* were not all created on the same display.

copy_area = **proc**(*w*: window, *p*: pixmap, *gc*: gcontext, *srcx*, *srcy*, *width*, *height*, *destx*, *desty*: int)
signals (*display_conflict*)
requires *w*.depth = *p*.depth.
modifies *w*.
effects Copies *p* onto *w* using *gc*. Only the portion of *p* which begins at (*srcx*, *srcy*) and has dimensions *width* and *height* is copied; that portion is placed on *w* with its upper-left corner at (*destx*, *desty*). Signals *display_conflict* if *w*, *p*, and *gc* were not all created on the same display.

end window

pixmap = **data type is** make_none, is_none, make_parent_relative, is_parent_relative, is_normal, create, load_bitmap, load_gif, destroy, equal, get_display, get_dimensions, get_width, get_height, get_depth, draw_point, draw_line, draw_rectangle, fill_rectangle, draw_lines, fill_polygon, draw_arc, fill_arc, draw_string, draw_image_string, copy_plane, copy_area

A pixmap is a (generally off-screen) drawing area. Windows are also pixmaps; the window$w2p (page 8) operation creates a pixmap which is equivalent to a given window.

Aside from actual pixmaps, there are two special pixmap objects which are useful for some calls: the *none* object and the *parent_relative* object. All calls with a pixmap parameter require that the pixmap is an actual pixmap object, not *none* or *parent_relative*, unless otherwise noted. All calls also require that any pixmap parameters have not been destroyed; see destroy, page 11.

make_none = **proc**(*d*: display) **returns** (pixmap)
effects Returns a *none* pixmap for the given display.

is_none = **proc**(*p*: pixmap) **returns** (bool)
requires *p* may be any pixmap object.
effects Returns true if and only if *p* is a *none* pixmap.

make_parent_relative = **proc**(*d*: display) **returns** (pixmap)
effects Returns a *parent_relative* pixmap for the given display.

is_parent_relative = **proc**(*p*: pixmap) **returns** (bool)
requires *p* may be any pixmap object.
effects Returns true if and only if *p* is a *parent_relative* pixmap.

is_normal = **proc**(*p*: pixmap) **returns** (bool)
requires *p* may be any pixmap object.
effects Returns true if and only if *p* is an actual pixmap object, not *none* or *parent_relative*.

create = **proc**(*reference*: pixmap, *width*, *height*, *depth*: int) **returns** (pixmap)
effects Creates and returns a pixmap of the given width, height, and depth. *Reference* is used to determine the display and some other information about the created pixmap.

load_bitmap = **proc**(*reference*: pixmap, *file*: string) **returns** (pixmap) **signals** (*unreadable*, *invalid*, *no_memory*)
effects Creates a pixmap of depth 1, containing a picture specified by the file named *file*, which must be in XBM (X11 Bitmap) format, and returns it. *Reference* is used to determine the display and some other information about the created pixmap. Signals *unreadable* if *file* is unreadable, *invalid* if it is not in XBM format, and *no_memory* if there is not enough memory to create the pixmap.

load_gif = **proc**(*reference*: pixmap, *file*: string) **returns** (pixmap) **signals** (*cannot*)
effects Creates a pixmap whose depth is the same as *reference*'s, containing a picture specified by the file named *file*, which must be in GIF 1987 format, and returns it. *Reference* is used to determine the display and depth of, and some other information about, the created pixmap. Signals *cannot* if the pixmap cannot be created for any reason.

destroy = **proc**(*p*: pixmap)
modifies *p*.
effects Destroys the given pixmap. All memory associated with *p* is freed; also, any pixmap objects referring to it become invalid, and cannot be used. If a pixmap still in use – that is, the background pixmap of some window, or the tile of some gcontext, etc. – is destroyed, heinous problems will rampage.

equal = **proc**(*p1*, *p2*: pixmap) **returns** (bool)
requires *p1* and *p2* may be any pixmap objects.
effects Returns true if and only if *p1* and *p2* refer to the same pixmap.

get_display = **proc**(*p*: pixmap) **returns** (display)
effects Returns the display *p* was created on.

get_dimensions = **proc**(*p*: pixmap) **returns** (int, int)
get_width = **proc**(*p*: pixmap) **returns** (int)
get_height = **proc**(*p*: pixmap) **returns** (int)
effects Return the dimensions of the pixmap *p*. Get_dimensions returns both width and height, in that order.

get_depth = **proc**(*p*: pixmap) **returns** (int)
effects Returns the depth of the pixmap *p*. A total of 2^{depth} different colors can be used on *p*.

The drawing operations for pixmaps are exactly the same as the drawing operations for windows, only with the first window argument, *w*, replaced by a pixmap, *p*. They are all described in more detail under their variants in the window cluster.

draw_point = **proc**(*p*: pixmap, *gc*: gcontext, *x*, *y*: int) **signals** (*display_conflict*)
See window$draw_point, page 8.

draw_line = **proc**(*p*: pixmap, *gc*: gcontext, *x1*, *y1*, *x2*, *y2*: int) **signals** (*display_conflict*)
See window$draw_line, page 8.

draw_rectangle = **proc**(*p*: pixmap, *gc*: gcontext, *x*, *y*, *width*, *height*: int) **signals** (*display_conflict*)
See window$draw_rectangle, page 8.

fill_rectangle = **proc**(*p*: pixmap, *gc*: gcontext, *x*, *y*, *width*, *height*: int) **signals** (*display_conflict*)
See window$fill_rectangle, page 8.

draw_lines = **proc**(*p*: pixmap, *gc*: gcontext, *points*: array[point]) **signals** (*display_conflict*)
See window$draw_lines, page 9.

fill_polygon = **proc**(*p*: pixmap, *gc*: gcontext, *points*: array[point]) **signals** (*display_conflict*)
See window$fill_polygon, page 9.

draw_arc = **proc**(*p*: pixmap, *gc*: gcontext, *x*, *y*, *width*, *height*: int, *angle1*, *angle2*: real)
signals (*display_conflict*)
See window$draw_arc, page 9.

fill_arc = **proc**(*p*: pixmap, *gc*: gcontext, *x*, *y*, *width*, *height*: int, *angle1*, *angle2*: real) **signals** (*display_conflict*)
See window$fill_arc, page 9.

draw_string = **proc**(*p*: pixmap, *gc*: gcontext, *x*, *y*: int, *s*: string) **signals** (*display_conflict*)
draw_image_string = **proc**(*p*: pixmap, *gc*: gcontext, *x*, *y*: int, *s*: string) **signals** (*display_conflict*)
See window$draw_string, page 9 and window$draw_image_string, page 9.

copy_plane = **proc**(*p*, *sourcep*: pixmap, *gc*: gcontext, *srcx*, *srcy*, *width*, *height*, *destx*, *desty*, *plane*: int)
signals (*display_conflict*)
See window$copy_plane, page 9.

copy_area = **proc**(*p*, *sourcep*: pixmap, *gc*: gcontext, *srcx*, *srcy*, *width*, *height*, *destx*, *desty*: int)
signals (*display_conflict*)
See window$copy_area, page 9.

end pixmap

gcontext = **data type is** create, destroy, equal, get_display, set_foreground, get_foreground, set_background, get_background, set_font, get_font, set_function, get_function, set_line_width, get_line_width, set_cap_style, get_cap_style, set_join_style, get_join_style, set_line_style, get_line_style, set_dashes, set_fill_rule, get_fill_rule, set_fill_style, get_fill_style, set_tile, get_tile, set_stipple, get_stipple, set_arc_mode, get_arc_mode, set_clip_mask, set_clip_mask_origin

A gcontext, short for graphics context, is just that: a context in which graphics routines are executed. A gcontext specifies, for all drawing routines, the foreground color, the background color, the way pixels are painted, whether lines are dashed, etc.; but it does no drawing on its own.

All routines which have gcontext parameters require that the gcontext has not been destroyed; see destroy, page 12. Also, drawing routines in the pixmap and window clusters which take gcontexts as parameters require that the thing being drawn into – the pixmap or window – has the same depth as the window the gcontext was created with.

create = **proc**(*w*: window) **returns** (gcontext)
effects Creates and returns a gcontext with the same display and depth as *w*. The gcontext can be used with windows and pixmaps other than *w*, but all windows or pixmaps it is used on must have the same display and depth as *w* did.

destroy = **proc**(*gc*: gcontext)
effects Destroys the given gcontext. All memory associated with *gc* is freed; also, any gcontext objects referring to it become invalid, and cannot be used.

equal = **proc**(*gc1*, *gc2*: gcontext) **returns** (bool)
effects Returns true if and only if *gc1* and *gc2* refer to the same gcontext.

get_display = **proc**(*gc*: gcontext) **returns** (display)
effects Returns the display which *gc* was created on.

set_foreground = **proc**(*gc*: gcontext, *c*: color) **signals** (*display_conflict*)
get_foreground = **proc**(*gc*: gcontext) **returns** (color)
modifies Set_foreground modifies *gc*.
effects Sets or gets *gc*'s foreground color. Set_foreground signals *display_conflict* if *gc* and *c* were not created on the same display.

set_background = **proc**(*gc*: gcontext, *c*: color) **signals** (*display_conflict*)
get_background = **proc**(*gc*: gcontext) **returns** (color)
modifies Set_background modifies *gc*.
effects Sets or gets *gc*'s background color. Set_background signals *display_conflict* if *gc* and *c* were not created on the same display.

set_font = **proc**(*gc*: gcontext, *f*: font) **signals** (*display_conflict*)
get_font = **proc**(*gc*: gcontext) **returns** (font)
modifies Set_font modifies *gc*.
effects Sets or gets *gc*'s font; that is, the font to be used when *gc* is used to draw text. Set_font signals *display_conflict* if *gc* and *f* were not created on the same display.

gcontext_function = **oneof**[clear, and, and_reverse, copy, and_inverted, noop, xor, nor, equiv, invert, or_reverse, copy_inverted, or_inverted, nand, set: null]
set_function = **proc**(*gc*: gcontext, *f*: gcontext_function)
get_function = **proc**(*gc*: gcontext) **returns** (gcontext_function)
modifies Set_function modifies *gc*.
effects Sets or gets *gc*'s transfer function, which specifies how pixels already present on a window or pixmap affect what happens when you draw.

set_line_width = **proc**(*gc*: gcontext, *lw*: int)
get_line_width = **proc**(*gc*: gcontext) **returns** (int)
requires $lw \geq 0$
modifies Set_line_width modifies *gc*.
effects Sets or gets *gc*'s line width, which is in pixels. In set_line_width, an *lw* of 0 tells X to draw very thin lines using a possibly faster algorithm than an *lw* of 1.

gcontext_cap_style = **oneof**[not_last, butt, round, projecting: null]
set_cap_style = **proc**(*gc*: gcontext, *capstyle*: gcontext_cap_style)
get_cap_style = **proc**(*gc*: gcontext) **returns** (gcontext_cap_style)
modifies Set_cap_style modifies *gc*.
effects Sets or gets *gc*'s cap style; that is, how the endpoints of a line are drawn. A *capstyle* of not_last means draw all but the last pixel in the line; butt means draw everything in the line; and round and projecting mean continue the line beyond its endpoint by one-half the current line width: round means to fill that space with a semicircle, projecting means to fill that space with a rectangle.

gcontext_join_style = **oneof**[miter, round, bevel: null]
set_join_style = **proc**(*gc*: gcontext, *joinstyle*: gcontext_join_style)
get_join_style = **proc**(*gc*: gcontext) **returns** (gcontext_join_style)
modifies Set_join_style modifies *gc*.
effects Sets or gets *gc*'s join style; that is, how two joined line segments (like you get in window$draw_lines (page 9)) are connected. A *joinstyle* of miter means to miter the corner – continue the edges of the two segments until they meet; round means to fill the space of the join with a filled circle; and bevel means to fill the triangular notch left by the join.

gcontext_line_style = **oneof**[solid, on_off_dash, double_dash: null]
set_line_style = **proc**(*gc*: gcontext, *linestyle*: gcontext_line_style)
get_line_style = **proc**(*gc*: gcontext) **returns** (gcontext_line_style)
modifies Set_line_style modifies *gc*.
effects Sets or gets *gc*'s line style; that is, whether lines are drawn solid (*linestyle* is solid) or dashed, and if dashed, whether the dashes are drawn by alternating foreground color and no drawing whatsoever (*linestyle* is on_off_dash) or by alternating between foreground and background colors (*linestyle* is double_dash).

set_dashes = **proc**(*gc*: gcontext, *offset*: int, *dashlist*: array[int])
requires For all $x \in$ *dashlist*, $x > 0$.
modifies *gc*.
effects Sets *gc*'s dash list and dash offset. The *dashlist* contains the lengths, in pixels, of each component of a dashed line. For example, if *dashlist* was [4, 2, 4, 8], a dashed line would be drawn 4 pixels on, 2 off, 4 on, and 8 off, repeated as often as necessary. If *dashlist* has an odd number of elements, it is repeated twice to form a full dash pattern; for example [4, 2, 3] means 4 on, 2 off, 3 on, 4 off, 2 on, 3 off. *Offset* specifies the number of pixels from the beginning of the dash list at which a line begins.

gcontext_fill_rule = **oneof**[even_odd, winding_number: null]
set_fill_rule = **proc**(*gc*: gcontext, *fillrule*: gcontext_fill_rule)
get_fill_rule = **proc**(*gc*: gcontext) **returns** (gcontext_fill_rule)
modifies Set_fill_rule modifies *gc*.
effects Sets or gets *gc*'s fill rule; that is, whether polygons are filled using the non-zero winding number rule (*fillrule* is winding_number), or the even-odd rule (*fillrule* is even_odd).

gcontext_fill_style = **oneof**[solid, tiled, stippled, opaque_stippled: null]
set_fill_style = **proc**(*gc*: gcontext, *fillstyle*: gcontext_fill_style)
get_fill_style = **proc**(*gc*: gcontext) **returns** (gcontext_fill_style)
modifies Set_fill_style modifies *gc*.
effects Sets or gets *gc*'s fill style; that is, whether filled shapes are filled with the foreground color (*fillstyle* is solid), the current tile (*fillstyle* is tiled), the current foreground color but only for pixels on in the current stipple (*fillstyle* is stippled), or the current foreground and background color, depending on the current stipple (*fillstyle* is opaque_stippled).

set_tile = **proc**(*gc*: gcontext, *tile*: pixmap) **signals** (*display_conflict*)
get_tile = **proc**(*gc*: gcontext) **returns** (pixmap)
requires Set_tile requires that *tile*.depth = *gc*'s depth.
modifies Set_tile modifies *gc*.
effects Sets or gets *gc*'s tile, which is a pixmap used to fill filled shapes when *gc*'s fill-style is tiled. Set_tile signals *display_conflict* if *gc* and *tile* were created on different displays.

set_stipple = **proc**(*gc*: gcontext, *stipple*: pixmap) **signals** (*display_conflict*, *wrong_depth*)
get_stipple = **proc**(*gc*: gcontext) **returns** (pixmap)
modifies Set_stipple modifies *gc*.
effects Sets or gets *gc*'s stipple, which is a bitmap – a one-plane pixmap – used to fill filled shapes when *gc*'s fill-style is stippled or opaque_stippled. Set_stipple signals *display_conflict* if *gc* and *stipple* were created on different displays, or *wrong_depth* if *stipple* has more than one plane.

set_arc_mode = **proc**(*gc*: gcontext, *arcmode*: gcontext_arc_mode)
get_arc_mode = **proc**(*gc*: gcontext) **returns** (gcontext_arc_mode)
modifies Set_arc_mode modifies *gc*.
effects Sets or gets *gc*'s arc mode; that is, whether filled arcs are filled like slices of a pie (*arcmode* is pie_slice) or marked off with a chord from the first point of the arc to the last (*arcmode* is chord).

set_clip_mask = **proc**(*gc*: gcontext, *clipmask*: pixmap) **signals** (*display_conflict*, *wrong_depth*)
modifies *gc*.
effects Sets *gc*'s clipping mask to *clipmask*, which must be a bitmap – a one-plane pixmap. In any drawing request, a pixel will be drawn only if it falls inside the boundaries of the clipping mask, and on an on (1) pixel there. See also set_clip_mask_origin, below. Signals *display_conflict* if *gc* and *clipmask* were not created on the same display, and *wrong_depth* if *clipmask* has more than one plane.

set_clip_mask_origin = **proc**(*gc*: gcontext, *xorigin*, *yorigin*: int)
modifies *gc*.
effects Sets the origin of *gc*'s clipping mask to (*xorigin*, *yorigin*). This means that the upper left corner of the clipping mask will be considered to be at (*xorigin*, *yorigin*), relative to the upper-left corner of whatever is being drawn into. See set_clip_mask, above.

end gcontext

cursor = **data type is** make_none, is_none, is_normal, create, create_from_pixmaps, destroy, equal, recolor

A cursor object visibly represents the position of the mouse pointer on the screen. You can set the pointer's appearance when it's in a certain window with window$set_cursor (page 6) and window$unset_cursor (page 6).

Aside from actual cursors, there is a special kind of cursor object which is useful for some operations: the *none* object. All calls with a cursor parameter require that the cursor is an actual cursor object, not *none*, unless otherwise noted. All calls also require that any cursor parameters have not been destroyed; see destroy, page 16.

make_none = **proc**(*d*: display) **returns** (cursor)
effects Creates and returns a *none* cursor for the given display.

is_none = **proc**(*c*: cursor) **returns** (bool)
requires *c* may be any cursor object.
effects Returns true if and only if *c* is a *none* cursor.

is_normal = **proc**(*c*: cursor) **returns** (bool)
requires *c* may be any cursor object.
effects Returns true if and only if *c* is an actual cursor object, not *none*.

create = **proc**(*d*: display, *shape*: int) **returns** (cursor)
requires *shape* is the offset for a valid cursor in the cursor font.
effects Creates and returns a cursor object from the *shape*th cursor in the cursor font. To look at the available cursors from the cursor font, run 'xfd -fn cursor'. To find out what number represents a given cursor shape, just click on the first of the two similar shapes; xfd will say something like 'character 0x0058 (0,88)'. The '88' is the number of that cursor shape.

create_from_pixmaps = **proc**(*d*: display, *source*, *mask*: pixmap, *xhot*, *yhot*: int) **returns** (cursor)
signals (*display_conflict*, *size_mismatch*, *wrong_depth*)
effects Creates and returns a cursor object from two bitmaps, a source bitmap and a mask bitmap. The hot spot – the point of the cursor that represents the position of the mouse – is located at (*xhot*, *yhot*) relative to the upper-left corner of *source*. When the cursor is drawn, first *mask*'s on pixels are drawn in the cursor's background color, then *source*'s on pixels are drawn in the cursor's foreground color. Signals *display_conflict* if *source* and *mask* were not both created on *d*, *size_mismatch* if *source* and *mask* have different sizes, and *wrong_depth* if *source* and *mask* do not both have depth 1.

destroy = **proc**(*c*: cursor)
modifies *c*.
effects Destroys the given cursor. All memory associated with *c* is freed; also, any cursor objects referring to it become invalid, and cannot be used. If a cursor still associated with a window is destroyed, you will be in hale, and you will not be able to come back from your dead lines.

equal = **proc**(*c1*, *c2*: cursor) **returns** (bool)
effects Returns true if and only if *c1* and *c2* refer to the same cursor.

recolor = **proc**(*c*: cursor, *fg*, *bg*: color)
modifies *c*.
effects Sets *c*'s foreground color to *fg*, and background color to *bg*. Note that *c*, *fg*, and *bg* need not have been created on the same display.

end cursor

color = **data type is** is_mutable, create, create_mutable, parse, destroy, equal, get_red, get_green, get_blue, get_rgb, set_red, set_green, set_blue, change, set_parsed

A color object represents a color on the screen. Colors can be mutable or immutable, and are created by two different functions (create and create_mutable) depending. In general, it is better to create immutable colors, as immutable colors can be shared between different X programs, which saves colormap space; but certain animation effects, like color cycling, are only possible with mutable colors.

All calls require that any color parameters have not been destroyed; see destroy, page 17.

is_mutable = **proc**(*c*: color) **returns** (bool)
effects Returns true if and only if *c* is mutable.

create = **proc**(*w*: window, *red*, *green*, *blue*: int) **returns** (color) **signals** (*colormap_full*)
requires $0 \leq$ *red,green,blue* < 65536.
effects Returns an immutable color usable on the window *w* and specified by the three numbers *red*, *green*, and *blue*. The returned color is actually usable on any window or pixmap with the same colormap as *w*, and the same display as *w*. (In the current incarnation of Clux, all windows with the same display have the same colormap.) Signals *colormap_full* if the color could not be allocated because the colormap is full.

create_mutable = **proc**(*w*: window, *red*, *green*, *blue*: int) **returns** (color) **signals** (*colormap_full*)
requires $0 \leq$ *red,green,blue* < 65536.
effects Returns a mutable color usable on the window *w* and specified by the three numbers *red*, *green*, and *blue*. The returned color is actually usable on any window or pixmap with the same colormap as *w*, and the same display as *w*. (In the current incarnation of Clux, all windows with the same display have the same colormap.) Signals *colormap_full* if the color could not be allocated because the colormap is full.

parse = **proc**(*w*: window, *name*: string) **returns** (color) **signals** (*not_defined*, *colormap_full*)
effects Returns an immutable color usable on the window *w* and specified by the string *name*. For example, passing "red" to color$parse will return a color which looks red on the screen. The returned color is actually usable on any window or pixmap with the same colormap as *w*, and the same display as *w*. (In the current incarnation of Clux, all windows with the same display have the same colormap.) Color names are looked up in the X11 color database (usually /usr/lib/X11/rgb.txt); in addition, the 'number syntax', "#RGB", "#RRGGBB", etc., is supported to specify colors by hexadecimal RGB values. Signals *not_defined* if *name* does not specify a valid color, and *colormap_full* if the color could not be allocated because the colormap is full.

destroy = **proc**(*c*: color)
effects Deallocates the color *c*, making the colormap space it took up available for other programs to use. It's a good idea to do this as soon as you're done with a color.

equal = **proc**(*c1*, *c2*: color) **returns** (bool)
effects Returns true if and only if *c1* and *c2* refer to the same color.

get_red = **proc**(*c*: color) **returns** (int)
get_green = **proc**(*c*: color) **returns** (int)
get_blue = **proc**(*c*: color) **returns** (int)
get_rgb = **proc**(*c*: color) **returns** (int, int, int)
effects Return the red, green, or blue component (or all three, in that order) of *c* as an integer between 0 and 65535, where 0 is black.

set_red = **proc**(*c*: color, *red*: int) **signals** (*immutable*)
set_green = **proc**(*c*: color, *green*: int) **signals** (*immutable*)
set_blue = **proc**(*c*: color, *blue*: int) **signals** (*immutable*)
change = **proc**(*c*: color, *red*, *green*, *blue*: int) **signals** (*immutable*)
requires $0 \leq$ *red,green,blue* < 65536
effects Set the red, green, or blue component (or all three) of *c*. Signals *immutable* if *c* is immutable.

set_parsed = **proc**(*c*: color, *name*: string) **signals** (*immutable*, *not_defined*)
effects Sets *c*'s red, green, and blue components to values specified by the color name *name* (see parse, page 17). Signals *immutable* if *c* is immutable.

end color

font = **data type is** parse, destroy, equal, names, text_width, text_extents

A font specifies how to draw a string; that is, what shape will correspond to the letter 'a', etc.

All calls with a font argument require that that font has not been destroyed; see destroy, page 18.

parse = **proc**(*d*: display, *s*: string) **returns** (font) **signals** (*nonexistent*)
effects Loads the font specified by *s* and returns it. *S* should be the name of a valid font; it can contain '*' characters, which act as wildcards: one '*' in *s* can stand for any number of characters in the "correct" font name. Signals *nonexistent* if a font named *s* does not exist on the display *d*.

destroy = **proc**(*f*: font)
modifies *f*.
effects Destroys the given font. All memory associated with *f* is freed; also, any font objects referring to it become invalid, and cannot be used. If a font still associated with a gcontext is destroyed, you will graduate in 2 years with a 5.0 GPA and a triple major.

equal = **proc**(*f1*, *f2*: font) **returns** (bool)
effects Returns true if and only if *f1* and *f2* refer to the same font.

names = **iter**(*d*: display, *pattern*: string) **yields** (string)
effects Yields, in no particular order, the name of every font which exists on the display *d* whose name matches the pattern *pattern*, which can contain '*' wildcards (see parse, page 18).

text_width = **proc**(*f*: font, *s*: string) **returns** (int)
effects Returns the width the string *s* would be if it was drawn with the font *f*.

text_extents = **proc**(*f*: font, *s*: string) **returns** (int, int)
effects Returns the ascent and descent the string *s* would have if it was drawn with the font *f*, in that order.

end font

flag_set = **data type is** new, test, equal, set, add, clear, sub

A flag_set is a set of integer flags, with operations to add or subtract flags from the set and to test whether a flag is in the set. It is used in Clux to implement event masks (see event_mask, page 20) and modifier sets (see modifier_set, page 23). The integer flags must be powers of 2 for the flag_set to work correctly.

new = **proc**() **returns** (flag_set)
effects Creates and returns a new flag set with no flags on.

flag = int
test = **proc**(*f*: flag_set, *fl*: flag) **returns** (bool)
effects Returns true if and only if the flag *fl* is on in *f*.

equal = **proc**(*f1*, *f2*: flag_set) **returns** (bool)
effects Returns true if and only if *f1* and *f2* are equal flag_sets. Two flag_sets are equal when a flag is on in *f1* if and only if it is on in *f2*.

set = **proc**(*f*: flag_set, *fl*: flag)
modifies *f*.
effects Modifies *f* so that *fl* is on in *f*.

add = **proc**(*f*: flag_set, *fl*: flag) **returns** (flag_set)
effects Returns a new flag set, *f2*, so that *f* is identical to *f2* except that *fl* is on in *f2*. (If *fl* was already on in *f*, *f2* and *f* are identical.)

clear = **proc**(*f*: flag_set, *fl*: flag)
modifies *f*.
effects Modifies *f* so that *fl* is off in *f*.

sub = **proc**(*f*: flag_set, *fl*: flag) **returns** (flag_set)
effects Returns a new flag set, *f2*, so that *f* is identical to *f2* except that *fl* is off in *f2*. (If *fl* was already off in *f*, *f2* and *f* are identical.)

end flag_set

event: A oneof, containing many species of event.

Events are the mechanism which Clux uses to notify your program of the user's actions, and things like when your windows become viewable or unviewable. To receive events, you first must select the events to be received for your application's windows, using an event_mask (page 20) and window$select_input (page 6). Then calls to operations like window$mask_event (page 7) or display$check_typed_event (page 4) return events to you as they happen.

There are many different species of event, each with different specifications, fields, etc. Thus, the event-returning operations return a oneof, event, which can be any one of the species of event. Some species are currently unimplemented (X has an awful lot of events, and most of them are not important to most X applications); those have a value type of 'null'.

You normally won't create an event yourself. However, the operations display$check_typed_event (page 4) and window$check_typed_event (page 7) each take an event, and will return the next matching event with the same type. To create an event from scratch to pass to these functions, use either 'event$make_*species*(nil)' if *species* has a null value type or 'event$make_*species*(*type-of-species*$make_null())' if *species*'s value type isn't null.

Events are selected with event masks, which are special flag_sets. An event can only be received if it has been selected for the window it applies to with window$select_input (page 6). (Note that window$create_toplevel (page 5) performs an implicit window$select_input.) Some event retrieving commands, like window$mask_event (page 7), also take an event mask to specify what kind of events will be returned. See event_mask, page 20.

```
event = oneof[
    button_press: button_event,          % Returned when a mouse button is pressed
    button_release: button_event,        % Returned when a mouse button is released
    enter_notify: crossing_event,        % Returned when the pointer enters a window
    leave_notify: crossing_event,        % Returned when the pointer leaves a window
    motion_notify: motion_event,         % Returned when the pointer moves
    key_press: key_event,                % Returned when a key is pressed
    key_release: key_event,              % Returned when a key is released
    expose: expose_event,                % Returned when a previously obscured portion
                                         %   of a window becomes visible
    map_notify: map_event,               % Returned when a window is mapped
    unmap_notify: map_event,             % Returned when a window is unmapped
    configure_notify: configure_event,   % Returned when a window changes size or position
    focus_in: focus_event,               % Returned when a window gets the keyboard focus
    focus_out: focus_event,              % Returned when a window loses the keyboard focus
    (and many null events)
]
```

event_mask: A special flag_set which selects events.

An event_mask is used to specify what subset of events a window (see window$select_input, page 6) or a particular event-returning call (see display$mask_event, page 4, for example) is interested in. See event, page 19, for a description of what the different event species are.

event_mask = flag_set (see page 18)

To create an event mask, use event_mask$new (see flag_set$new, page 18), which creates a new event_mask with no events selected. Then, use event_mask$add (see flag_set$add, page 19) to add the event_masks you want on to it. For example, 'event_mask$new() + key_press_mask + leave_window_mask' creates an event_mask with key_press and leave_notify event species selected.

Note that, even if you don't select any events, some event species can always be returned to you, no matter what. They all have null types, and, for some, the Clux system internally deals with them in the correct manner. You can ignore them safely.

This event mask	*Selects these event species*
no_event_mask	(no events)
key_press_mask	key_press
key_release_mask	key_release
button_press_mask	button_press
button_release_mask	button_release
enter_window_mask	enter_notify
leave_window_mask	leave_notify
pointer_motion_mask	motion_notify
pointer_motion_hint_mask	(This selects no events on its own, but causes any motion_notify events returned to be hints. See motion_event$get_is_hint, page 25.)
exposure_mask	expose
structure_notify_mask	map_notify, unmap_notify, configure_notify
focus_change_mask	focus_in, focus_out

key_event = **data type is** make_null, get_display, get_window, get_time, get_mouse, get_mouse_x, get_mouse_y, get_key, get_key_char, get_modifiers

A key_event is an event returned when the user presses or releases a key. It is the type of the key_press and key_release event species; see event, page 19.

make_null = **proc**() **returns** (key_event)
effects Returns a key_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder key_event whose contents are not relevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: key_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: key_event) **returns** (window)
effects Returns the window to which *e* applies: the window with the keyboard focus.

get_time = **proc**(*e*: key_event) **returns** (time)
effects Returns the time at which *e* occurred.

get_mouse = **proc**(*e*: key_event) **returns** (int, int)
get_mouse_x = **proc**(*e*: key_event) **returns** (int)
get_mouse_y = **proc**(*e*: key_event) **returns** (int)
effects Returns the position of the mouse when *e* occurred relative to *e*.window's upper-left corner. Get_mouse returns both x and y positions, in that order.

get_key = **proc**(*e*: key_event) **returns** (key)
effects Returns the key which was pressed or released.

get_key_char = **proc**(*e*: key_event) **returns** (char) **signals** (*no_char_value*)
effects Returns the character equivalent of the key which was pressed or released, or signals *no_char_value* if the key which was pressed has no character equivalent. For example, the '1' key has a character equivalent of '1', but 'Compose Character' has no character equivalent.

get_modifiers = **proc**(*e*: key_event) **returns** (modifier_set)
effects Returns the set of modifiers which were being pressed when *e* occurred.

end key_event

key = **data type is** equal, k2i, is_modifier, is_keypad_key, is_function_key, function_key_num, is_pf_key, pf_key_num

A key is an abstract representation of a key on a keyboard, which is more general than a simple character. Some key constants are provided for a subset of the keys which have no character values.

These key constants	*Represent these keys*
left_key, up_key, right_key, down_key	the arrow keys
delete_key	the Backspace key above Return on a DEC
backspace_key	(does not exist on DECs)
tab_key, return_key, enter_key	Tab, Return, Enter (Enter is on the keypad)
help_key, prev_key, next_key	Help, Prev, Next
left_shift_key, right_shift_key	the left and right Shift keys
left_control_key, right_control_key	the left and right Control keys (a right Control key doesn't exist on DECs)
left_meta_key, right_meta_key	the left and right Meta keys (do not exist on DECs)
left_alt_key, right_alt_key	the left and right Alt keys
compose_character_key	either Compose Character key

equal = **proc**(*k*: key, *i*: int) **returns** (bool)
effects Returns true if and only if *k* and the key represented by *i* are equal. This operation takes an int as its second argument to facilitate comparing keys with integer key constants like 'return_key': you can say '*k* = return_key' in programs, where *k* is a key.

k2i = **proc**(*k*: key) **returns** (int)
effects Returns the integer representation of *k*.

is_modifier = **proc**(*k*: key) **returns** (bool)
effects Returns true if and only if *k* is a modifier key, like Shift, Alt, or Control.

is_keypad_key = **proc**(*k*: key) **returns** (bool)
effects Returns true if and only if *k* is a key on the numeric keypad.

is_function_key = **proc**(*k*: key) **returns** (bool)
effects Returns true if and only if *k* is a function key (F1, F2, F3...).

function_key_num = **proc**(*k*: key) **returns** (int) **signals** (*not_function_key*)
effects Returns the number of the function key *k*; for example, if *k* represents F6, function_key_num returns 6. Signals *not_function_key* if *k* is not a function key.

is_pf_key = **proc**(*k*: key) **returns** (bool)
effects Returns true if and only if *k* is a PF key (PF1, PF2, ...).

pf_key_num = **proc**(*k*: key) **returns** (int) **signals** (*not_pf_key*)
effects Returns the number of the PF key *k*; for example, if *k* represents PF2, pf_key_num returns 2. Signals *not_pf_key* if *k* is not a PF key.

end key

modifier_set: A special flag_set which tells which modifiers are being pressed.

A modifier_set can be obtained from several event species (see button_event$get_modifiers, page 24 and crossing_event$get_modifiers, page 24, for example), and tells which modifiers were being held down at the time of the event. Mouse buttons are modifiers, as well as the modifier keys (keys like Shift, Control, and Alt).

modifier_set = flag_set (see page 18)

To test if a given modifier is in a modifier_set, use modifier_set$test (see flag_set$test, page 18) with one of the following modifier constants. Note that the binding of modifier constant to physical key or button can change; this is only the default setup, which is also the setup that nearly everyone has. (man xmodmap for details on how to change the modifier/physical key binding.)

This modifier constant	*Is the...*
shift_modifier	Shift key
lock_modifier	Shift Lock key
control_modifier	Control key
mod1_modifier	Meta key (it's labeled Compose Character on DECs)
mod2_modifier	Alt key, on DECs
mod3_modifier	(not bound to any key by default)
mod4_modifier	(not bound to any key by default)
mod5_modifier	(not bound to any key by default)
button1_modifier	left mouse button
button2_modifier	middle mouse button
button3_modifier	right mouse button
button4_modifier	(Athena mice have only three buttons)
button5_modifier	(Athena mice have only three buttons)

button_event = **data type is** make_null, get_display, get_window, get_time, get_mouse, get_mouse_x, get_mouse_y, get_button, get_modifiers

A button_event is an event returned when the user clicks a mouse button in a window. It is the type of the button_press and button_release event species; see event, page 19.

make_null = **proc**() **returns** (button_event)
effects Returns a button_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder button_event whose contents are irrelevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: button_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: button_event) **returns** (window)
effects Returns the window the pointer was in when *e* occurred.

time = int
get_time = **proc**(*e*: button_event) **returns** (time)
effects Returns the time at which *e* occurred.

get_mouse = **proc**(*e*: button_event) **returns** (int, int)
get_mouse_x = **proc**(*e*: button_event) **returns** (int)
get_mouse_y = **proc**(*e*: button_event) **returns** (int)
effects Returns the position of the mouse when *e* occurred relative to *e*.window's upper-left corner. Get_mouse returns both x and y positions, in that order.

get_button = **proc**(*e*: button_event) **returns** (int)
effects Returns the button number of the button that caused *e*.

get_modifiers = **proc**(*e*: button_event) **returns** (modifier_set)
effects Returns the set of modifiers which were on when *e* occurred.

end button_event

crossing_event = **data type is** make_null, get_display, get_window, get_time, get_mouse, get_mouse_x, get_mouse_y, get_focus, get_modifiers

A crossing_event is an event returned when the pointer either enters or leaves a window. It is the type of the enter_notify and leave_notify event species; see event, page 19.

make_null = **proc**() **returns** (crossing_event)
effects Returns a crossing_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder crossing_event whose contents are irrelevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: crossing_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: crossing_event) **returns** (window)
effects Returns the window which was entered or left.

get_time = **proc**(*e*: crossing_event) **returns** (time)
effects Returns the time at which *e* occurred.

get_mouse = **proc**(*e*: crossing_event) **returns** (int, int)
get_mouse_x = **proc**(*e*: crossing_event) **returns** (int)
get_mouse_y = **proc**(*e*: crossing_event) **returns** (int)
effects Return the position of the mouse when *e* occurred relative to *e*.window's upper-left corner. Get_mouse returns both the x and y positions, in that order.

get_focus = **proc**(*e*: crossing_event) **returns** (bool)
effects Returns true if and only if *e*.window, or one of *e*.window's parents, now has the keyboard focus.

get_modifiers = **proc**(*e*: crossing_event) **returns** (modifier_set)
effects Returns the set of modifiers which were on when *e* occurred.

end crossing_event

motion_event = **data type is** make_null, get_display, get_window, get_time, get_mouse, get_mouse_x, get_mouse_y, get_modifiers, get_is_hint

A motion_event is an event returned when the user moves the pointer. It is the type of the motion_notify event species; see event, page 19.

make_null = **proc**() **returns** (motion_event)
effects Returns a motion_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder motion_event whose contents are irrelevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: motion_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: motion_event) **returns** (window)
effects Returns the window the pointer was in when *e* occurred.

get_time = **proc**(*e*: motion_event) **returns** (time)
effects Returns the time at which *e* occurred.

get_mouse = **proc**(*e*: motion_event) **returns** (int, int)
get_mouse_x = **proc**(*e*: motion_event) **returns** (int)
get_mouse_y = **proc**(*e*: motion_event) **returns** (int)
effects Return the position of the mouse when *e* occurred relative to *e*.window's upper-left corner. Get_mouse returns both the x and y positions, in that order.

get_modifiers = **proc**(*e*: motion_event) **returns** (modifier_set)
effects Returns the set of modifiers which were on when *e* occurred.

get_is_hint = **proc**(*e*: motion_event) **returns** (bool)
effects Returns true if and only if this motion event is a hint: that is, it is not a true reflection of where the pointer is now; the application should use display$query_pointer (page 4) to find out where the pointer is.

end motion_event

map_event = **data type is** make_null, get_display, get_window

A map_event is an event returned when a window is mapped or unmapped. It is the type of the map_notify and unmap_notify event species; see event, page 19.

make_null = **proc**() **returns** (map_event)
effects Returns a map_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder map_event whose contents are irrelevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: map_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: map_event) **returns** (window)
effects Returns the window which was mapped or unmapped.

end map_event

expose_event = **data type is** make_null, get_display, get_window, get_expose_count, get_exposed_area, get_exposed_area_x, get_exposed_area_y, get_exposed_area_width, get_exposed_area_height

An expose_event is an event returned when a portion of a window which was invisible becomes visible. It is the type of the expose event species; see event, page 19.

make_null = **proc**() **returns** (expose_event)
effects Returns a expose_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder expose_event whose contents are irrelevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: expose_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: expose_event) **returns** (window)
effects Returns the window which was exposed.

get_expose_count = **proc**(*e*: expose_event) **returns** (int)
effects Returns the expose count; that is, the number of remaining expose events which were generated for *e*.window by the circumstances which generated *e*. If your application reacts to an expose event by redrawing the entire window, you may want to check the expose_count field, and only redraw if *e*.expose_count = 0. If *e*.expose_count > 0, you know that at least *e*.expose_count events remain to be processed, so you might as well ignore this one.

get_exposed_area = **proc**(*e*: expose_event) **returns** (int, int, int, int)
get_exposed_area_x = **proc**(*e*: expose_event) **returns** (int)
get_exposed_area_y = **proc**(*e*: expose_event) **returns** (int)
get_exposed_area_width = **proc**(*e*: expose_event) **returns** (int)
get_exposed_area_height = **proc**(*e*: expose_event) **returns** (int)
effects Return the position and/or dimensions of the rectangular area exposed by *e*. Get_exposed_area returns x, y, width, and height, in that order.

end expose_event

configure_event = **data type is** make_null, get_display, get_window, get_position_x, get_position_y, get_width, get_height, get_border_width

A configure_event is an event returned when a window moves or changes size for some reason. It is the type of the configure_notify event species; see event, page 19.

make_null = **proc**() **returns** (configure_event)
effects Returns a configure_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder configure_event whose contents are irrelevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: configure_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: configure_event) **returns** (window)
effects Returns the window which changed size.

get_position_x = **proc**(*e*: configure_event) **returns** (int)
get_position_y = **proc**(*e*: configure_event) **returns** (int)
effects Return the new position of *e*.window relative to its parent.

get_width = **proc**(*e*: configure_event) **returns** (int)
get_height = **proc**(*e*: configure_event) **returns** (int)
effects Return the new dimensions of *e*.window.

get_border_width = **proc**(*e*: configure_event) **returns** (int)
effects Returns the new border width of *e*.window.

end configure_event

focus_event = **data type is** make_null, get_display, get_window, get_mode

A focus_event is an event returned when a window either loses or gains the keyboard focus. It is the type of the focus_in and focus_out event species; see event, page 19.

make_null = **proc**() **returns** (focus_event)
effects Returns a focus_event whose fields are all garbage. Performing any 'get' functions on this event will cause trouble. Useful mainly for creating a placeholder focus_event whose contents are irrelevant, for instance in calls to display$check_typed_event (page 4) and window$check_typed_event (page 7).

get_display = **proc**(*e*: focus_event) **returns** (display)
effects Returns *e*'s associated display.

get_window = **proc**(*e*: focus_event) **returns** (window)
effects Returns the window which gained or lost focus.

focus_event_mode = **oneof**[normal, grab, ungrab: null]
get_mode = **proc**(*e*: focus_event) **returns** (focus_event_mode)
effects Returns the mode with which the keyboard focus was stolen. If the focus event happened because the keyboard was grabbed, the focus_event_mode will be grab; if it happened when the keyboard was ungrabbed, focus_event_mode will be ungrab; and if it happened for any other reason, focus_event_mode will be normal.

end focus_event

resource_database = **data type is** create, create_file, destroy, equal, merge, fetch, store, store_line

A resource_database controls the fast storage and retrieval of resources, which are bits of data categorized by *specifications*. Resources are retrieved by two specifications, and stored with only one. A specification consists of one or more *components*, each of which is a string not containing any of the characters '.', '*', or ':'. The components are separated from one another by *bindings*, which are either *tight* bindings, represented by a dot '.', or *loose* bindings, represented by a star '*'. When retrieving a resource, the type of binding doesn't matter; however, when storing a resource, the loose binding serves as sort of a componentwise wildcard: it can stand for any number of missing components, and thus makes one stored resource able to satisfy requests for many different retrieved resources. For example, if a resource database contains a specification 'xterm*thingie: 100', requests for the resources 'xterm.thingie', 'xterm.this.thingie', and 'xterm.another.confusing.thingie' will all return the '100' specified by the 'xterm*thingie' resource.

When you retrieve a resource, you give two specifications: the *name* specification and the *class* specification. Class specifications are supposed to be more general than name specifications; they might indicate an entire category of resources, rather than one specific resource. A stored specification can contain mixed class and name components and still match. However, a stored specification which matches all name components will take precedence over a stored specification which matches all class components.

The most common use for a resource_database is to retrieve preferences set by the user in their ~/.Xdefaults or ~/.Xresources file; see display$get_resource_database, page 3.

All calls require that any resource_database parameters have not been destroyed; see resource_database$destroy, page 28.

create = **proc**(*s*: string) **returns** (resource_database)
effects Creates and returns a resource database containing only the stored specifications listed in *s*. *S* should consist of a number of lines, separated by newline characters, and each line should have first a specification, then a colon ':' character, then some optional spaces which are ignored, then the value to be stored for that specification. (To continue a value onto multiple lines, use '\n' to stand for a newline.) To create an empty database, pass an empty string to create. Lines beginning with an exclamation point '!' are treated as comments and ignored.

create_file = **proc**(*fn*: string) **returns** (resource_database)
effects Creates and returns a resource database containing only the stored specifications listed in the file whose name is *fn*. If *fn* does not name a valid file, returns an empty database. The file should satisfy the conditions listed above at create; that is, specifications are separated by newlines, '!' lines are comments, etc.

destroy = **proc**(*rdb*: resource_database)
modifies *rdb*.
effects Frees all the memory associated with *rdb*.

equal = **proc**(*rdb1*, *rdb2*: resource_database) **returns** (bool)
effects Returns true if and only if *rdb1* and *rdb2* refer to the same resource_database.

merge = **proc**(*mergeinto*, *mergesource*: resource_database)
modifies *mergeinto*.
effects Adds all the specifications in *mergesource* to *mergeinto*. The specifications taken from *mergesource* take precedence over existing specifications in *mergeinto*.

fetch = **proc**(*rdb*: resource_database, *namestr*, *classstr*: string) **returns** (string) **signals** (*nonexistent*)
effects Returns the value stored in *rdb* for the retrieval specifications *namestr* (the name components) and *classstr* (the class components), or signals *nonexistent* if no such value is in the database.

store = **proc**(*rdb*: resource_database, *specification*, *value*: string)
modifies *rdb*.
effects Stores the string *value* in *rdb* with the storage specification *specification*.

store_line = **proc**(*rdb*: resource_database, *line*: string)
modifies *rdb*.
effects Stores the line *line*, which should be in the format of one specification as accepted by resource_database$create (page 28), into *rdb*. That is, *line* should contain the specification, followed by a ':', followed by some optional spaces, followed by the value.

end resource_database