CLX
Common LISP X Interface

ACKNOWLEDGMENTS

Primary Interface Author:
Robert W. Scheifler
MIT Laboratory for Computer Science
545 Technology Square, Room 418
Cambridge, MA 02139
rws@zermatt.lcs.mit.edu

Primary Implementation Author:
LaMott Oren
Texas Instruments
PO Box 655474, MS 238
Dallas, TX 75265
oren@csc.ti.com

Design Contributors:
Dan Cerys, BBN
Scott Fahlman, CMU
Kerry Kimbrough, Texas Instruments
Chris Lindblad, MIT
Rob MacLachlan, CMU
Mike McMahon, Symbolics
David Moon, Symbolics
LaMott Oren, Texas Instruments
Daniel Weinreb, Symbolics
John Wroclawski, MIT
Richard Zippel, Symbolics

Documentation Contributors:
Keith Cessna, Texas Instruments
Kerry Kimbrough, Texas Instruments
Mike Myjak
LaMott Oren, Texas Instruments
Dan Stenger, Texas Instruments

CONTENTS

INTRODUCTION TO CLX

1

Introduction

1.1 This manual assumes a basic understanding of window systems and the Common Lisp programming language. To provide an introduction to the Common Lisp X Interface (CLX) programming, this section discusses the following:

- Overview of the X Window System
- Naming and argument conventions
- Programming considerations

The X Window System

1.2 The X Window System was developed at the Massachusetts Institute of Technology (MIT) and first released in 1985. Since then, the X Window System has become an industry-standard product available on virtually every type of bit-mapped workstation. The current version of X, Version 11, has been implemented for several different computer architectures, for a wide variety of display hardware, and also for many different operating systems. X Version 11 represents the fulfillment of the original design goals proposed by MIT, as follows:

- Portable — Support virtually any bitmap display and any interactive input device (including keyboards, mice, tablets, joysticks, and touch screens). Make it easy to implement the window system on different operating systems.
- Device-Independent Applications — Avoid rewriting, recompiling, or even relinking in order to use different display/input hardware. Make it easy for an application to work on both monochrome and color hardware.
- Network Transparent — Let an application run on one computer while using another computer's display, even if the other computer has a different operating system or hardware architecture.
- Multitasking — Support multiple applications being displayed simultaneously.
- No User Interface Policy — Since no one agrees on what constitutes the best user interface, make it possible for a broad range of user interface styles (or policies) to be implemented, external to the window system and to the application programs.
- Cheap Windows — Windows should be abundant, and ubiquitous. Provide overlapping windows and a simple mechanism for window hierarchy.
- High-Performance Graphics — Provide powerful interfaces for synthesizing 2-D images (geometric primitives, high-quality text with multiple typefaces, and scanned images).
- Extensible — Include a mechanism for adding new capabilities. Allow separate sites to develop independent extensions without becoming incompatible with remote applications.

Some of these goals lead directly to the basic X architecture — the client-server model. The basic window system is implemented by the X *server* program. An application program (the *client*) sends window system *requests* to the X server through a reliable two-way byte-stream.

In general, the server and the client can be executing on separate host computers, in which case the byte-stream is implemented via some network protocol (TCP, DECnet™, Chaosnet, and so forth). The X server, which is connected to several client programs running concurrently, executes client requests in round-robin fashion. The server is responsible for drawing client graphics on the display screen and for making sure that graphics output to a window stays inside its boundary.

The other primary job of the X server is to channel input from the keyboard, pointer, and other input devices back to the appropriate client programs. Input arrives at the client asynchronously in the form of input *events* representing up/down transitions of keys or pointer buttons, changes in the pointer position, and so on. In some cases, a request generates a return value (or *reply*) from the server, which is another kind of client input. Replies and input events are received via the same byte-stream connecting the client with the server.

Windows

1.2.1 The X Window System supports one or more screens containing overlapping windows and subwindows. A *screen* is a physical monitor and hardware, which can be either color or black and white. There can be multiple screens per display workstation. A single server can provide display services for any number of screens. A set of screens for a single user with one keyboard and one mouse is called a *display*.

All windows in an X server are arranged in a strict hierarchy. At the top of the hierarchy are the *root windows*, which cover each of the display screens. Each root window is either partially or completely covered by child windows. All windows, except for root windows, have parents. Any window can in turn have its own children. In this way, an application program can create a window tree of arbitrary depth on each screen.

A child window can be larger than its parent. That is, part or all of the child window can extend beyond the boundaries of the parent. However, all output to a window is clipped by the boundaries of its parent window. If several children of a window have overlapping locations, one of the children is considered to be on top of/or raised over the others, *obscuring* them. Window output to areas that are covered by other windows is suppressed.

A window has a border that is zero or more pixels in width and can be any pattern (pixmap) or solid color. A window usually has a background pattern that is drawn by the X server. Each window has its own coordinate system. Child windows obscure their parents unless the child windows have no background. Graphics operations in the parent window are usually clipped by the children.

X also provides objects called *pixmaps* for off-screen storage of graphics. Single-plane pixmaps (that is, of depth 1) are sometimes referred to as *bitmaps*. Both pixmaps and windows can be used interchangeably in most graphics functions. Pixmaps are also used in various graphics operations to define patterns, or *tiles*. Windows and pixmaps together are referred to as *drawables*.

Input Events

1.2.2 The X input mechanism is conceptually simple yet quite powerful. Most events are attached to a particular window (that is, contain an identifier for the window receiving the event). A client program can receive multiple window input streams, all multiplexed over the single byte-stream connection to the server.

Clients can tailor their input by expressing interest in only certain event types. The server uses special event types to send important messages to the client. For example, the client can elect to receive an **:enter-notify** event when the pointer cursor moves into a certain window. Another vital message from the server is an **:exposure** event. This is a signal to the client indicating that at least some portion of the window has suddenly become visible (perhaps the user moved another window which had been overlapping it). The client is then responsible for doing what is necessary to redisplay the window's image. Client programs must be prepared to regenerate the contents of windows in this way on demand.

Input is also subject to policy decisions about which client window receives keyboard and pointer events. Since the pointer is free to roam between windows, just clicking on a window is often enough to send a pointer event to that window. Keyboard events, however, must go to a keyboard focus window which has to be designated in some other way. Usually, the arbiter of such input management policy is a program called the *window manager*. The window manager gives the human user a way to make a window the keyboard focus, to manage the layout of windows on the screen, to represent windows with icons, and so forth. In fact, the window manager client determines most of the so-called look and feel of the X Window System.

A Quick Tour of CLX

1.3 The X Window System is defined by the X Window System Protocol Specification, a detailed description of the encoding and the meaning of requests and events sent between a client and a server. This standard protocol does not depend on any particular programming language. As a result, each programming language must define its own functional interface for using the X protocol. The standard X interface used by Common Lisp programmers is called CLX. CLX is a set of data types, functions, and macros which allow a Common Lisp client program to interact with an X server to send requests and to receive input events and replies.

For the most part, CLX functions are closely tied to the underlying requests in the X protocol. Many CLX functions simply add requests to an output buffer. These requests later execute asynchronously on the X display server. However, some functions of CLX lie outside the scope of the protocol—for example, reading events and managing a client-side event queue. CLX is also responsible for important batching and caching tasks that minimize network communication.

The following paragraphs show an example of a CLX client program. All CLX functions and macros are shown in upper case. Note that some of the terms used are unique to X, while other terms that are common to other window systems have different meanings in X. It may be helpful to refer to the glossary when you are uncertain of a term's meaning in the context of the X Window System.

A Simple Menu

1.3.1 The example client program creates and displays a simple pop-up menu consisting of a column of strings—a title string followed by selectable menu item strings. The implementation uses one window to represent the entire menu, plus a set of subwindows, one for each menu item. Here is the definition of a structure which represents such a menu.

```
(defstruct (menu)
  "A simple menu of text strings."
  (title "Choose an item:")
  item-alist              ;((item-window item-string))
  window
  gcontext
  width
  title-width
  item-width
  item-height
  (geometry-changed-p t)) ;nil if unchanged since displayed
```

The `window` slot will contain the **window** object that represents the menu. The `item-alist` represents the relationship between the menu items and their associated subwindows. Each entry in `item-alist` is a list whose first element is a (sub)window object and whose second element is the corresponding item string. A **window** object is an instance of a CLX-defined data type which represents X windows. A **window** object actually carries two pieces of information: an X window ID integer and a **display** object. A **display** is another CLX-defined data type that represents a connection to a specific X display server. The `gcontext` slot contains an instance of a CLX data type known as a *graphics context*. A graphics context is a set of display attribute values, such as foreground color, fill style, line style, text font, and so forth. Each X graphics request (and hence each CLX graphics function call) must supply a graphics context to use in displaying the request. The menu's `gcontext` will thus hold all of the attribute values used during menu display.

The first thing to do is make an instance of a `menu` object:

```
(defun create-menu (parent-window text-color background-color
 text-font)
  (make-menu
    ;; Create menu graphics context
    :gcontext (CREATE-GCONTEXT :drawable   parent-window
                               :foreground text-color
                               :background background-color
                               :font       text-font)

    ;; Create menu window
    :window   (CREATE-WINDOW
               :parent       parent-window
               :class        :input-output
               :x            0                 ;temporary value
               :y            0                 ;temporary value
               :width        16                ;temporary value
               :height       16                ;temporary value
               :border-width 2
               :border       text-color
               :background   background-color
               :save-under   :on
               :override-redirect :on          ;override window mgr when positioning
               :event-mask   (MAKE-EVENT-MASK :leave-window

                                              :exposure))))
```

create-window is one of the most important CLX functions, since it creates and returns a **window** object. Several of its options are shown here. The default window class is **:input-output**, but X provides for **:input-only** windows, too. Every window must have a parent window, except for a system-defined *root window*, which represents an entire display screen. The **:event-mask** keyword value, a CLX **event-mask** data type, says that an input event will be received for the menu window when the window is exposed and also when the pointer cursor leaves the window. The window border is a pattern-filled or (as in this case) a solid-colored boundary which is maintained automatically by the X server; a client cannot draw in a window's border, since all graphics requests are relative to the origin (upper-left corner) of the window's interior and are clipped by the server to this inside region. Turning on the **:save-under** option is a hint to the X server that, when this window is made visible, it may be more efficient to save the pixels it obscures, rather than require several client programs to refresh their windows when the pop-up menu disappears. This is a way to work around X's client-managed refresh policy when only a small amount of screen space is needed temporarily.

Why is **:override-redirect** turned on for the menu window? This is actually a little unusual, because it prevents any window manager client from *redirecting* the position of the menu when it is popped up. Remember that the window manager represents the user's policy for controlling the positions of his windows, so this kind of redirection is ordinarily correct. However, in this case, as a favor to the user, the menu avoids redirection in order to pop up the menu at a very specific location; that is, under the pointer cursor.

What about the item subwindows? The `menu-set-item-list` function in the following example creates them whenever the menu's item list is changed. The upper-left x and y coordinates and the width and height are not important yet, because they are computed just before the menu is displayed. This function also calls **create-window**, demonstrating the equal treatment of parent and children windows in the X window hierarchy.

```
(defun menu-set-item-list (menu &rest item-strings)
  ;; Assume the new items will change the menu's width and height
  (setf (menu-geometry-changed-p menu) t)

  ;; Destroy any existing item windows
  (dolist (item (menu-item-alist menu))
    (DESTROY-WINDOW (first item)))

  ;; Add (item-window item-string) elements to item-alist
  (setf (menu-item-alist menu)
        (let (alist)
          (dolist (item item-strings (nreverse alist))
            (push (list (CREATE-WINDOW
                          :parent     (menu-window menu)
                          :x          0         ;temporary value
                          :y          0         ;temporary value
                          :width      16        ;temporary value
                          :height     16        ;temporary value
                          :background (GCONTEXT-BACKGROUND (menu-gcontext menu))
                          :event-mask (MAKE-EVENT-MASK :enter-window
                                                       :leave-window
                                                       :button-press
                                                       :button-release))
                        item)
                  alist)))))
```

Displaying the Menu

1.3.2 The `menu-recompute-geometry` function (shown in the following example) handles the job of calculating the size of the menu, based on its current item list and its current text font. CLX provides a way to inquire the geometrical properties of a font object (for example, its ascent and descent from the baseline) and also a **text-extents** function. **text-extents** returns the geometry of a given string as displayed in a given font. Notice the use of the **with-state** macro when setting a window's geometry attributes. CLX strives to preserve the familiar **setf** style of accessing individual window attributes, even though an attribute access actually involves sending a request to a (possibly remote) server and/or waiting for a reply. **with-state** tells CLX to batch together all read and write accesses to a given window, using a local cache to minimize the number of server requests. This CLX feature can result in a dramatic improvement in client performance without burdening the programmer interface.

`menu-recompute-geometry` causes all the item subwindows to become *mapped*. Mapping a window means attempting to make it visible on the screen. However, a subwindow will not actually be *visible* until it and all of its ancestors are mapped. Even then, another window might be covering up the subwindow.

```
(defun menu-recompute-geometry (menu)
  (when (menu-geometry-changed-p menu)
    (let* ((menu-font   (GCONTEXT-FONT (menu-gcontext menu)))
           (title-width (TEXT-EXTENTS menu-font (menu-title menu)))
           (item-height (+ (FONT-ASCENT menu-font)
                           (FONT-DESCENT menu-font)
                           *menu-item-margin*))
           (item-width  0)
           (items       (menu-item-alist menu))
           menu-width)

     ;; Find max item string width
     (setf item-width
           (+ *menu-item-margin*
              (dolist (next-item items item-width)
                (setf item-width (max item-width
                                      (TEXT-EXTENTS menu-font (second next-item)))))))

     ;; Compute final menu width, taking margins into account
     (setf menu-width (max title-width (+ item-width *menu-item-margin*)))
     (let ((window  (menu-window menu)))

     ;; Update width and height of menu window
     (WITH-STATE (window)
       (setf (DRAWABLE-WIDTH  window) menu-width
             (DRAWABLE-HEIGHT window) (* (1+ (length items)) item-height)))

     ;; Update width, height, position of item windows
     (let ((item-left     (round (- menu-width item-width) 2))
           (next-item-top (- item-height (round *menu-item-margin* 2))))
       (dolist (next-item items)
         (let ((window (first next-item)))
           (WITH-STATE (window)
             (setf (DRAWABLE-HEIGHT window) item-height
                   (DRAWABLE-WIDTH  window) item-width
                   (DRAWABLE-X      window) item-left
                   (DRAWABLE-Y      window) next-item-top)))
         (incf next-item-top item-height))))

     ;; Map all item windows
     (MAP-SUBWINDOWS (menu-window menu))

     ;; Save item geometry
     (setf (menu-item-width menu)         item-width
           (menu-item-height menu)        item-height
           (menu-width menu)              menu-width
           (menu-title-width menu)        title-width
           (menu-geometry-changed-p menu) nil))))
```

Of course, the sample client must know how to draw/redraw the menu and its items, so the function `menu-refresh` is defined next to handle that task (shown in the following example). Note that the location of window output is given relative to the window origin. Windows and subwindows have different coordinate systems. The location of the origin (upper-left corner) of a subwindow's coordinate system is given with respect to its parent window's coordinate system. Negative coordinates are valid, although only output to the +x/+y quadrant of a window's coordinate system will ever be visible.

```
(defun menu-refresh (menu)
 (let* ((gcontext   (menu-gcontext menu))
        (baseline-y (FONT-ASCENT (GCONTEXT-FONT gcontext))))
   ;; Show title centered in "reverse-video"
   (let ((fg (GCONTEXT-BACKGROUND gcontext))
         (bg (GCONTEXT-FOREGROUND gcontext)))
     (WITH-GCONTEXT (gcontext :foreground fg :background bg)
       (DRAW-IMAGE-GLYPHS
         (menu-window menu)
         gcontext
         (round (- (menu-width menu)
                   (menu-title-width menu)) 2)  ;start x
         baseline-y                             ;start y
         (menu-title menu))))

   ;; Show each menu item (position is relative to item window)
   (let ((box-margin (round *menu-item-margin*  2)))
     (dolist (item (menu-item-alist menu))
       (DRAW-IMAGE-GLYPHS
         (first item) gcontext
         box-margin                             ;start x
         (+ baseline-y box-margin)              ;start y
         (second item))))))
```

with-gcontext is a CLX macro that allows you temporarily to modify a graphics context within the dynamic scope of the macro body. **draw-image-glyphs** is a CLX text drawing function which produces a terminal-like rendering: foreground character on a background block. (More sophisticated text rendering functions are also available.) The strange use of *glyphs* instead of *string* here actually highlights an important fact: X and Common Lisp have totally different concepts of a character. A Common Lisp character is an object whose implementation can comprehend a vast universe of text complexities (typefaces, type styles, international character sets, symbols, and so forth). However, to X, a string is just a sequence of integer indexes into the array of bitmaps represented by a CLX font object. In general, **draw-image-glyphs**, **text-extents**, and other CLX text functions accept a **:translate** keyword argument. Its value is a function which translates the characters of a string argument into the appropriate font-and-index pairs needed by CLX. This example relies upon the default translation function, which simply uses **char-code** to compute an index into the current font.

Menu Input

1.3.3 Now that a menu can be displayed, the sample client program must define how the menu will process user input. The `menu-choose` function (shown in the following example) has the classic structure of an X client program. First, do some initialization (for example, present the menu at a given location). Then, enter an input event loop. Read an input event, process it, and repeat the loop until a termination event is received. The **event-case** macro continues reading an event from the menu window's display object until one of its clauses returns non-**nil**. These clauses specify the action to be taken for each event type and also bind values from the event report to local variables, such as the **event-window** receiving the event. Notice that the **:force-output-p** option is enabled, causing **event-case** to begin by sending any client requests which CLX has not yet output to the server. To improve performance, CLX quietly queues up requests and periodically sends them off in a batch. However, in an interactive feedback loop such as this, it is important to keep the display crisply up-to-date.

```
(defun menu-choose (menu x y)
  ;; Display the menu so that first item is at x,y.
  (menu-present menu x y)

   (let ((items (menu-item-alist menu))
         (mw    (menu-window menu))
         selected-item)

    ;; Event processing loop
    (do () (selected-item)
      (EVENT-CASE ((DRAWABLE-DISPLAY mw) :force-output-p t)
        (:exposure
          (count)
          ;; Discard all but final :exposure then display the menu
          (when (zerop count) (menu-refresh menu))
          t)

        (:button-release
          (event-window)
          ;;Select an item
          (setf selected-item (second (assoc event-window items)))
          t)

        (:enter-notify
          (window)
          ;;Highlight an item
          (menu-highlight-item menu (find window items :key #'first))
          t)

        (:leave-notify
          (window kind)
          (if (eql mw window)
              ;; Quit if pointer moved out of main menu window
              (setf selected-item (when (eq kind :ancestor) :none))
              ;; Otherwise, unhighlight the item window left
              (menu-unhighlight-item menu (find window items :key #'first)))
            t)

          (otherwise
            ()
            ;;Ignore and discard any other event
            t)))

        ;; Erase the menu
        (UNMAP-WINDOW mw)

        ;; Return selected item string, if any
        (unless (eq selected-item :none) selected-item)))
```

The event loop in `menu-choose` demonstrates an idiom used in all X programs: the contents of a window are displayed (in this case, by calling `menu-refresh`) only when an **:exposure** event is received, signaling that the server has actually made the window *viewable*. The handling of **:exposure** in `menu-choose` also implements a little trick for improving efficiency. In general, when a window is exposed after being previously obscured (perhaps only partially), the server is free to send several **:exposure** events, one for each rectangular tile of the exposed region. For small windows like this menu, it is not worth the trouble to redraw the image one tile at a time. So the code above just ignores all but the last tile exposure and redraws everything in one call to `menu-refresh`.

The Main Program

1.3.4 After all the preceding build-up and the other functions referenced (but not shown here) have been implemented, the code for the main client program is very small.

```
(defun just-say-lisp (host &optional (font-name "fg-16"))
  (let* ((display   (OPEN-DISPLAY host))
         (screen    (first (DISPLAY-ROOTS display)))
         (fg-color  (SCREEN-BLACK-PIXEL screen))
         (bg-color  (SCREEN-WHITE-PIXEL screen))
         (nice-font (OPEN-FONT display font-name))

         ;; Create a menu as a child of the root window.
         (a-menu    (create-menu (SCREEN-ROOT screen)
                                 fg-color bg-color nice-font)))

    (setf (menu-title a-menu) "Please pick your favorite language:")
    (menu-set-item-list a-menu "Fortran" "APL" "Forth" "Lisp")

    ;; Bedevil the user until he picks a nice programming language
    (unwind-protect
        (loop
          ;; Determine the current root window position of the pointer
          (multiple-value-bind (x y) (QUERY-POINTER (SCREEN-ROOT screen))

           (let ((choice (menu-choose a-menu x y)))
             (when (string-equal "Lisp" choice)
               (return)))))


     (CLOSE-DISPLAY display))))
```

Note that the main program event loop lies in the body of an **unwind-protect** form. This is a good programming technique because, without this protection, an unexpected error could cause the program to terminate without freeing the *server resources* it has created. Server resources are CLX objects which refer to objects actually stored on the X server. Examples of these are **window**, **font**, **pixmap**, **cursor**, **colormap**, and **gcontext** objects. These server resources are created and destroyed by user requests. Server resources created by a client are also destroyed when its display connection is closed. If client resources are repeatedly created without being destroyed, then the server will eventually run out of memory and fail.

Most server resources are potentially sharable between applications. In fact, windows are manipulated explicitly by window manager programs. Fonts and cursors are typically shared automatically since the X server loads and unloads font storage as needed. **gcontext** objects are not ordinarily shared between client applications.

Debugging With CLX

1.3.5 Typically, most CLX programs do not need to control the buffering of output requests directly. However, CLX programmers need to be aware of the asynchronous nature of client-server communication. It may be convenient to control the CLX output buffer more directly, especially during debugging.

A client that wants a request to execute immediately instead of asynchronously can follow it with a call to **display-force-output**. This function *blocks* (does not return) until all previously buffered output requests have been sent. Otherwise, the output buffer is always flushed by a call to any function which returns a value from the server or which waits for input (for example, **get-property**). Certain output requests can cause input events to be sent. For example, **map-window** can cause **:exposure** events to be sent. Synchronizing output with the resulting input can be done with the **display-finish-output** function. This function blocks until all previously buffered output has been sent and all resulting input events have been received.

Functions that return information from the server block until an explicit reply is received or an error occurs. If a nonblocking call results in an error, the error is generally not reported until later. All errors (synchronous and asynchronous) are processed by calling an error handler defined for the display. If the handler is a sequence it is expected to contain handler functions specific to each error. The error code is used to index the sequence, fetching the appropriate handler. Any results returned by the handler are ignored since it is assumed that the handler either takes care of the error completely, or else signals.

Naming and Argument Conventions

1.4 Throughout CLX, a number of conventions for naming and syntax of the CLX functions have been followed. These conventions are intended to make the syntax of the functions more predictable.

The major naming conventions are as follows:

- To better differentiate the CLX symbols from other symbols, they have all been placed in the package XLIB. External symbols have been explicitly exported.
- The *display* argument, where used, is always first in the argument list.
- All server resource objects, where used, occur at the beginning of the argument list, immediately after the display variable.
- When a graphics context (*gcontext*) is present together with another type of server resource (most commonly, a *drawable*), the graphics context occurs in the argument list after the other server resource. Drawables out rank all other server resources.
- Source arguments always precede the destination arguments in the argument list.
- The *x* argument always precedes the *y* argument in the argument list.
- The *width* argument always precedes the *height* argument in the argument list.
- Where the *x*, *y*, *width* and *height* arguments are used together, the *x* and *y* arguments always precede the *width* and *height* arguments.
- Where a *mask* is accompanied with a *structure*, the mask always precedes the structure in the argument list.

Programming Considerations

1.5 The major programming considerations are as follows:

- Keyboards are the greatest variable between different manufacturer's workstations. If you want your program to be portable, you should be particularly conservative here.
- Many display systems have limited amounts of off-screen memory. If you can, you should minimize use of pixmaps and backing store.
- The user should have control of his screen real-estate. Therefore, you should write your applications to react to window management, rather than presume control of the entire screen. What you do inside of your top level window, however, is up to your application.
- Coordinates and sizes in X are actually 16-bit quantities. They usually are declared as an **int16** in the functions. Values larger than 16 bits can be truncated silently. Sizes (width and height) are unsigned quantities.

- The types **color**, **colormap**, **cursor**, **display**, **font**, **gcontext**, **pixmap**, **screen**, and **window** are defined solely by a functional interface. Even though they are treated like structures in this document, it is not specified whether they are implemented as structures or classes. Although some interfaces are described as functions, they are not required to be defined using **defun.** (It is a requirement that they be functions as opposed to macros or special forms.)

Data Types

1.6 The following are some data type definitions that are commonly used in CLX function definitions.

alist (*key-type-and-name datum-type-and-name*) **'list** — Type

alist defines an association list. An association list is a sequence, containing zero or more repetitions of the given elements with each of the elements expressed as (*type name*).

angle ‘(**number** ,(* –2pi) ,(* 2pi)) — Type

angle defines an angle in units of radians and is bounded by (–2π) and (2π). Note that we are explicitly using a different angle representation than what is actually transmitted in the protocol.

arc-seq '(**repeat-seq** (**int16** *x*) (**int16** *y*) (**card16** *width*) (**card16** *height*) (**angle** *angle1*) (**angle** *angle2*)) — Type

arc-seq defines a six-tuple sequence of the form (*x*, *y*, *width*, *height*, *angle1*, *angle2*). The points *x* and *y* are signed, 16-bit quantities with a range from –32,768 to 32,767. The *width* and *height* values are unsigned, 16-bit quantities and range from 0 to 65,535. *angle1* and *angle2* are in units of radians, and bounded by (–2π) and (2π).

array-index ‘(**integer** 0 ,*array-dimension-limit*) — Type

array-index defines a type which is a subtype of the integers and can be used to describe all variables that can be array indices. The range is inclusive because start and end array index specifiers can be one (1) past the end.

bit-gravity ' (**member** *gravity**) — Type

A keyword that specifies which region of a window should be retained when the window is resized.

gravity — One of the following:

:center	**:north**	**:south**	**:static**
:east	**:north-east**	**:south-east**	**:west**
:forget	**:north-west**	**:south-west**	

If a window is reconfigured without changing its inside width or height, then the contents of the window moves with the window and are not lost. Otherwise, the contents of a resized window are either moved or lost, depending on its bit-gravity attribute. See **window-bit-gravity**, in paragraph 4.3, Window Attributes, for additional information.

bitmap '(**array bit** (* *)) — Type

Specifies a two-dimensional array of bits.

---

bitmap-format Structure

A structure that describes the storage format of a bitmap.

The **bitmap-format** structure contains slots for **unit**, **pad**, and **lsb-first-p**. The **unit** member indicates the unit of increments used to maintain the bitmap data. The units available for use are 8, 16, or 32 bits. The **pad** member indicates how many bits are needed to pad the left edge of the scan-line. The **lsb-first-p** member is a predicate which indicates the ordering of bits with the bitmap unit.

unit Slot of **bitmap-format**

Type: (**member** 8 16 32).

The size of a contiguous grouping of bits, which can be 8, 16, or 32. The default is 8.

pad Slot of **bitmap-format**

Type: (**member** 8 16 32).

The number of bits to left-pad the scan-line, which can be 8, 16, or 32. The default is 8.

lsb-first-p Slot of **bitmap-format**

Type: **boolean**.

A predicate indicating whether the least significant bit comes first (**true**) or not (**nil**).

boolean '(**or nil** (**not nil**)) Type

boolean defines a type which is all inclusive. It is used for variables that can take on a true (non-**nil**) or false (**nil**) value.

boole-constant '(**member** *value**) Type

boole–constant defines a type that is a set of the values associated with the 16 boolean operation-code constants for the Common Lisp language. It is used for the set of allowed source and destination combination functions in a graphics context.

value — One of the following:

boole-1	**boole-c1**	**boole-nand**	**boole-xor**
boole-2	**boole-c2**	**boole-nor**	
boole-and	**boole-clr**	**boole-orc1**	
boole-andc1	**boole-eqv**	**boole-orc2**	
boole-andc2	**boole-ior**	**boole-set**	

card8 '(**unsigned-byte** 8) Type

An unsigned integer value that is a maximum of eight bits long. This gives a number of this type a range from 0 to 255.

card16 '(**unsigned-byte** 16) Type

An unsigned integer value that is a maximum of 16 bits long. This gives a number of this type a range from 0 to 65,535.

card29 '(**unsigned-byte** 29) Type

An unsigned integer value that is a maximum of 29 bits long. This gives a number of this type a range from 0 to 536,870,911.

card32 '(**unsigned-byte** 32) Type

An unsigned integer value that is a maximum of 32 bits long. This gives a number of this type a range from 0 to 4,294,967,295.

color '(**satisfies color-p**) Type

A **color**. See paragraph 9.2, Color Functions, for additional information.

colormap '(**satisfies colormap-p**) Type

A **colormap**. See paragraph 9.3, Colormap Functions, for additional information.

cursor '(**satisfies cursor-p**) Type

A **cursor**. See Section 10, Cursors, for additional information.

device-event-mask '(**or mask32** (**list device-event-mask-class**)) Type

Provides a way to specify a set of bits for an event bitmask. Two ways of specifying the bits are allowed: by setting the event bits in a 32 bit mask, or by listing the keyword names of the device related event bits in a list.

device-event-mask-class ' (**member** *event**) Type

A keyword name, for a device related event, that corresponds to a particular bit in an event bitmask. The set of names is a subset of the names in the type **event-mask-class**.

event — One of the following:

:button-1-motion	**:button-motion**
:button-2-motion	**:button-press**
:button-3-motion	**:key-press**
:button-4-motion	**:key-release**
:button-5-motion	**:pointer-motion**

display '(**satisfies display-p**) Type

A connection to an X server. See Section 2, Displays, for additional information.

drawable '(**or window pixmap**) Type

Both **windows** and **pixmaps** can be used as sources and destinations in graphics operations. **windows** and **pixmaps** together are known as *drawables*. However, an **:input-only** window cannot be used as a source or destination in a graphics operation.

draw-direction '(**member :left-to-right :right-to-left**) Type

Defines a list of rotation directions for drawing arcs and fonts. **draw-direction** can have the values of **:left-to-right** or **:right-to-left**.

error-key '(**member** *error**) Type

Defines a list of all predefined errors. All errors (synchronous and asynchronous) are processed by calling an error handler in the display. The handler is called with the display as the first argument and the error-key as its second argument.

error — One of the following:

:access	**:drawable**	**:implementation**	**:value**
:alloc	**:font**	**:length**	**:window**
:atom	**:gcontext**	**:match**	
:colormap	**:id-choice**	**:name**	
:cursor	**:illegal-request**	**:pixmap**	

event-key '(**member** *event-type**) Type

Defines a list that specifies all predefined event-types. Clients are informed of information asynchronously by means of events. These events can be either asynchronously generated from devices or generated as side effects of client requests.

event-type — One of the following:

:button-press	**:exposure**	**:motion-notify**
:button-release	**:focus-in**	**:no-exposure**
:circulate-notify	**:focus-out**	**:property-notify**
:circulate-request	**:graphics-exposure**	**:reparent-notify**
:client-message	**:gravity-notify**	**:resize-request**
:colormap-notify	**:keymap-notify**	**:selection-clear**
:configure-notify	**:key-press**	**:selection-notify**
:configure-request	**:key-release**	**:selection-request**
:create-notify	**:leave-notify**	**:unmap-notify**
:destroy-notify	**:map-notify**	**:visibility-notify**
:enter-notify	**:map-request**	

event-mask '(**or mask32** (**list event-mask-class**)) Type

Provides a way to specify a set of bits for an event bitmask. Two ways of specifying the bits are allowed: by setting the event bits in a 32 bit mask, or by listing the keyword names of the event bits in a list.

---

event-mask-class ’ (**member** *event**) Type

The elements of the type **event-mask-class** are keyword names that correspond to a particular bit in an event bitmask.

event — One of the following:

:button-1-motion	**:enter-window**	**:pointer-motion-hint**
:button-2-motion	**:exposure**	**:property-change**
:button-3-motion	**:focus-change**	**:resize-redirect**
:button-4-motion	**:key-press**	**:structure-notify**
:button-5-motion	**:key-release**	**:substructure-notify**
:button-motion	**:keymap-state**	**:substructure-redirect**
:button-press	**:leave-window**	**:visibility-change**
:button-release	**:owner-grab-button**	
:colormap-change	**:pointer-motion**	

make-event-keys *event-mask* Function

Returns: *event-keywords* — Type **list**.

Returns a list of **event-mask-class** keyword names for the event bits that are set in the specified event mask.

event-mask — An event mask (type **mask32**).

make-event-mask &rest *keys* Function

Returns:
event-mask — Type **mask32**.

Constructs an event mask from a set of **event-mask-class** keyword names.

keys — **event-mask-class** keywords.

---

font ’(**satisfies font-p**) Type

A text font. See Section 8, Fonts and Characters, for additional information.

fontable ’(**or stringable font**) Type

A **fontable** is either a **font** object or the name of one of the fonts in the font database.

font-props ’**list** Type

A **list** that contains alternating keywords and integers.

gcontext ’(**satisfies gcontext-p**) Type

A graphics context. See Section 5, Graphics Contexts, for additional information.

gcontext-key ’(**member** *type**) Type

A list of predefined types for use in **gcontext** processing. Various information for graphics output is stored in a graphics context (GC or GContext), such as foreground pixel, background pixel, line width, clipping region, and so forth.

type — One of the following:

:arc-mode	**:exposures**	**:line-width**
:background	**:fill-rule**	**:plane–mask**
:cap-style :fill-style		**:stipple**
:clip-mask	**:font**	**:subwindow-mode**
:clip-x	**:foreground**	**:tile**
:clip-y	**:function**	**:ts-x**
:dash-offset	**:join-style**	**:ts-y**
:dashes	**:line-style**	

grab-status '(**member** *grab-type**) Type

There are two kinds of grabs: active and passive. An *active grab* occurs when a single client grabs the keyboard and/or pointer explicitly. Clients can also grab a particular keyboard key or pointer button in a window. The grab activates when the key or button is actually pressed, and is called a *passive grab*. Passive grabs can be very convenient for implementing reliable pop-up menus.

grab-type — One of the following:

:already-grabbed
:frozen
:invalid-time
:not-viewable
:success

image-depth '(**integer** 0 32) Type

Used in determining the depth of a pixmap, window, or image. The value specifies the number of bits deep that a given pixel has within a given pixmap, window, or image.

index-size '(**member :default** 8 16) Type

Used to control the element size of the destination buffer given to the translate function when drawing glyphs. If **:default** is specified, the size is based on the current font, if known; otherwise, 16 is used.

int8 '(**signed-byte** 8) Type

A signed integer value that is a maximum of eight bits long. A number of this type can have a range from –128 to 127.

int16 '(**signed-byte** 16) Type

A signed integer value that is a maximum of 16 bits long. A number of this type can have a range from –32,768 to 32,767.

int32 '(**signed-byte** 32) Type

A signed integer value that is a maximum of 32 bits long. A number of this type can have a range from –2,147,483,648 to 2,147,483,647.

keysym '**card32** Type

Used as an encoding of a symbol on a keycap on a keyboard. It is an unsigned integer value represented in a maximum of 32 bits long. A **keysym** type can have a range from 0 to 4,294,967,295.

mask16 ' **card16** Type

A positional bitmask that contains 16 boolean flags.

mask32 ’ **card32** Type

A positional bitmask that contains 32 boolean flags.

modifier-key ’(**member** *modifier****) Type

A keyword identifying one of the modifier keys on the keyboard device.

modifier — One of the following:

:shift	**:mod-2**
:lock	**:mod-3**
:control	**:mod-4**
:mod-1	**:mod-5**

modifier-mask ’(**or** (**member :any**) **mask16** (**list modifier-key**)) Type

A bitmask or list of keywords that specifies a set of modifier keys. The keyword **:any** is equivalent to any subset of modifier key.

pixarray ’(**or** (**array pixel** (* *))
(**array card16** (* *))
(**array card8** (* *))
(**array** (**unsigned-byte 4**) (* *))
(**array bit** (* *))) Type

Specifies a two-dimensional array of pixels.

pixel ’(**unsigned-byte** 32) Type

An unsigned integer value that is a maximum of 32 bits long. This gives a pixel type a value range from 0 to 4,294,967,295. Useful values are dependent on the class of colormap being used.

pixmap ’(**satisfies pixmap-p**) Type

A **pixmap**. See paragraph 4.8, Pixmaps, for additional information.

---

pixmap-format Structure

A structure that describes the storage format of a pixmap.

The **pixmap-format** structure contains slots for **depth**, **bits-per-pixel**, and **scanline-pad**. The **depth** member indicates the number of bit planes in the pixmap. The **bits-per-pixel** member indicates the number of bits used to represent a single pixel. For X, a pixel can be 1, 4, 8, 16, 24, or 32 bits wide. As for **bitmap-format**, the **scanline-pad** member indicates how many pixels are needed to pad the left edge of the scan-line.

depth Slot of **pixmap-format**

Type: **image-depth**.

The number of bit planes in the pixmap.

bits-per-pixel Slot of **pixmap-format**

Type: (**member** 1 4 8 16 24 32).

The number of consecutive bits used to encode a single pixel. The default is 8.

scanline-pad Slot of **pixmap-format**

Type: (**member** 8 16 32).

The number of bits to left-pad the scan-line, which can be 8, 16, or 32. The default is 8.

---

point-seq '(**repeat-seq** (**int16** *x*) (**int16** *y*)) Type

The **point-seq** type is used to define sequences of (*x*,*y*) pairs of points. The paired values are 16-bit, signed integer quantities. This gives the points in this type a range from –32,768 to 32,767.

pointer-event-mask '(**or mask32** (**list pointer-event-mask-class**)) Type

Provides a way to specify a set of bits for an event bitmask. Two ways of specifying the bits are allowed: by setting the event bits in a 32 bit mask, or by listing the keyword names of the pointer related event bits in a list.

pointer-event-mask-class ' (**member** *event**) Type

A keyword name, for a pointer related event, that corresponds to a particular bit in an event bitmask. The set of names is a subset of the names in the type **event-mask-class**.

event — One of the following:

:button-1-motion	**:button-motion**	**:leave-window**
:button-2-motion	**:button-press**	**:pointer-motion**
:button-3-motion	**:button-release**	**:pointer-motion-hint**
:button-4-motion	**:enter-window**	
:button-5-motion	**:keymap-state**	

rect-seq '(**repeat-seq** (**int16** *x*) (**int16** *y*) (**card16** *width*) (**card16** *height*)) Type

rect-seq defines a four-tuple sequence of the form (*x*, *y*, *width*, *height*). The points *x* and *y* are signed, 16-bit quantities with a range from –32,768 to 32,767. The *width* and *height* values are unsigned, 16-bit quantities and range from 0 to 65,535.

repeat-seq (&rest *elts*) '**sequence** Type

A subtype used to define repeating sequences.

resource-id ’**card29** Type

A numeric identifier that is assigned by the server to a server resource object.

rgb-val ’(**float** 0.0 1.0) Type

An **rgb-val** is a floating-point value between 0 and 1 that specifies a saturation for a red, green, or blue additive primary. The 0 value indicates no saturation and 1 indicates full saturation.

screen ’(**satisfies screen-p**) Type

A display screen. See Section 3, Screens, for further information.

seg-seq ’(**repeat-seq** (**int16** *x1*) (**int16** *y1*) (**int16** *x2*) (**int16** *y2*)) Type

Defines sequences of (*x1*, *y1*, *x2*, *y2*) sets of points. The point values are 16-bit, signed integer quantities. This gives the points in this type a range from –32,768 to 32,767.

---

state-mask-key ’(**or modifier-key** (**member** *button**) Type

A keyword identifying one of the display modifier keys or pointer buttons whose state is reported in device events.

button — One of the following:

:button-1	**:button-4**
:button-2	**:button-5**
:button-3	

make-state-keys *state-mask* Function

Returns:
state-keywords — Type **list**.

Returns a list of **state-mask-key** symbols corresponding to the *state-mask*. A symbol belongs to the returned list if, and only if, the corresponding *state-mask* bit is 1.

state-mask — A 16-bit mask of type **mask16**.

make-state-mask &rest *keys* Function

Returns:
mask — Type **mask16**.

Returns a 16-bit *mask* representing the given **state-mask-key** symbols. The returned *mask* contains a 1 bit for each keyword.

keys — A list of **state-mask-key** symbols.

---

stringable ’(**or string symbol**) Type

Used for naming something. This type can be either a string or a **symbol** whose **symbol-name** is used as the string containing the name. The case of the characters in the string is ignored when comparing stringables.

timestamp '(**or null card32**) Type

An encoding of a time. **nil** stands for the current time.

---

visual-info Structure

A structure that represents a visual type. The elements of this structure are **id**, **class**, **red-mask**, **green-mask**, **blue-mask**, **bits-per-rgb**, and **colormap-entries**.

id Slot of **visual-info**

Type: **card29**.

A unique identification number.

class Slot of **visual-info**

Type: (**member :direct-color :gray-scale :pseudo-color :static-color :static-gray :true-color**).

The class of the visual type.

red-mask, **green-mask**, **blue-mask** Slots of **visual-info**

Type: **pixel**.

The **red-mask**, **green-mask**, and **blue-mask** elements are only meaningful for the **:direct-color** and **:true-color** classes. Each mask has one contiguous set of bits with no intersections.

bits-per-rgb Slot of **visual-info**

Type: **card8**.

Specifies the log base 2 of the approximate number of distinct color values (individually) of red, green, and blue. Actual RGB values are unsigned 16-bit numbers.

colormap-entries Slot of **visual-info**

Type: **card16**.

Defines the number of available colormap entries in a newly created colormap. For **:direct-color** and **:true-color**, this is the size of an individual pixel subfield.

---

win-gravity '(**member** *gravity**) Type

A keyword that specifies how to reposition a window when its parent is resized.

gravity — One of the following:

:center	**:north-west**	**:static**
:east	**:south**	**:unmap**
:north	**:south-east**	**:west**
:north-east	**:south-west**	

If a parent window is reconfigured without changing its inside width or height, then all child windows move with the parent and are not changed. Otherwise, each child of the resized parent is moved, depending on the child's gravity attribute. See **window-gravity**, in paragraph 4.3, Window Attributes, for additional information.

window '(**satisfies window-p**) Type

A window. See Section 4, Windows and Pixmaps, for additional information.

xatom '(**or string symbol**) Type

A name that has been assigned a corresponding unique ID by the server. **xatoms** are used to identify properties, selections, and types defined in the X server. An **xatom** can be either a **string** or **symbol** whose **symbol-name** is used as the **xatom** name. The case of the characters in the string are significant when comparing **xatoms**.

DISPLAYS 2

Introduction

2.1 A particular X server, together with its screens and input devices, is called a *display*. The CLX **display** object contains all the information about the particular display and its screens, as well as the state that is needed to communicate with the display over a particular connection.

Before your program can use a display, you must establish a connection to the X server driving your display. Once you have established a connection, you then can use the CLX macros and functions discussed in this section to return information about the display. This section discusses how to:

- Open (connect) a display
- Obtain information about a display
- Access and change display attributes
- Close (disconnect) a display

Opening the Display

2.2 The **open-display** function is used to open a connection to an X server.

open-display *host* &key **:display :protocol** Function

Returns:
display — Type **display**.

Returns a **display** that serves as the connection to the X server and contains all the information about that X server.

host — Specifies the name of the *host* machine on which the server executes. A string must be acceptable as a *host*, but otherwise the possible types are not constrained and will likely be very system dependent.

:display — An integer that specifies which display device on the *host* should be used for this connection. This is needed since multiple displays can be controlled by a single X server. The default is display 0 (zero).

:protocol — A keyword argument that specifies which network protocol should be used for connecting to the server (for example, **:tcp**, **:dna**, or **:chaos**). The set of possible values and the default value are implementation specific.

Authorization, if any, is assumed to come from the environment. After a successful call to **open-display**, all screens on the display can be used by the client application.

Display Attributes

2.3 The complete set of display attributes is discussed in the following paragraphs.

display-authorization-data *display* Function

Returns:
authorization-data — Type **string**.

Returns the authorization data string for *display* that was transmitted to the server by **open-display** during connection setup. The data is specific to the particular authorization protocol that was used. The **display-authorization-name** function returns the protocol used.

display — A **display** object.

display-authorization-name *display* Function

Returns:
authorization-name — Type **string**.

Returns the authorization protocol namestring for *display* that was transmitted by **open-display** to the server during connection setup. The *authorization-name* indicates what authorization protocol the client expects the server to use. Specification of valid authorization mechanisms is not part of the X protocol. A server that implements a different protocol than the client expects, or a server that only implements the host-based mechanism, can simply ignore this information. If both name and data strings are empty, this is to be interpreted as "no explicit authorization."

display — A **display** object.

display-bitmap-format *display* Function

Returns:
bitmap-format — Type **bitmap-format**.

Returns the *bitmap-format* information for the specified *display.*

display — A **display** object.

display-byte-order *display* Function

Returns:
byte-order — Either **:lsbfirst** or **:msbfirst**.

Returns the *byte-order* to be employed in communication with the server for the given *display.* The possible values are as follows:

- **:lsbfirst** — Values are transmitted least significant byte first.
- **:msbfirst** — Values are transmitted most significant byte first.

Except where explicitly noted in the protocol, all 16-bit and 32-bit quantities sent by the client must be transmitted with this *byte-order*, and all 16-bit and 32-bit quantities returned by the server are transmitted with this *byte-order.*

display — A **display** object.

display-display *display* Function

Returns:
display-number — Type **integer**.

Returns the *display-number* for the host associated with *display.*

display — A **display** object.

display-error-handler *display* Function

Returns:
error-handler — Type **function** or **sequence**.

Returns and (with **setf**) sets the *error-handler* function for the given *display*. CLX calls (one of) the display error handler functions to handle server errors returned to the connection. The default error handler, **default-error-handler**, signals conditions as they occur. See Section 16, Errors, for a list of the conditions that CLX can signal. For more information about errors and error handling, refer to the section entitled Common Lisp Condition System in the *Lisp Reference* manual.

If the value of *error-handler* is a sequence, it is expected to contain a handler function for each specific error. The error code is used as an index into the sequence to fetch the appropriate handler function. If this element is a function, it is called for all errors. Any results returned by the handler are ignored since it is assumed the handler either takes care of the error completely or else signals. The arguments passed to the handler function are the **display** object, a symbol naming the type of error, and a set of keyword-value argument pairs that vary depending on the type of error. For all core errors, the keyword-value argument pairs are:

:current-sequence	**card16**
:major	**card8**
:minor	**card16**
:sequence	**card16**

For **colormap**, **cursor**, **drawable**, **font**, **gcontext**, **id-choice**, **pixmap**, and **window** errors, the keyword-value pairs are the core error pairs plus:

:resource-id	**card32**

For **:atom** errors, the keyword-value pairs are the core error pairs plus:

:atom-id	**card32**

For **:value** errors, the keyword-value pairs are the core error pairs plus:

:value	**card32**

display — A **display** object.

display-image-lsb-first-p *display* Function

Returns:
image-lsb-first-p — Type **boolean**.

Although the server is generally responsible for byte swapping communication data to match the client, images (pixmaps/bitmaps) are always transmitted and received in formats (including byte order) specified by the server. Within images for each scan-line unit in bitmaps or for each pixel value in pixmaps, the leftmost bit in the image as displayed on the screen is either the least or most significant bit in the unit. For the given *display*, **display-image-lsb-first-p** returns non-**nil** if the leftmost bit is the least significant bit; otherwise, it returns **nil**.

display — A **display** object.

display-keycode-range *display* Function

Returns:
min-keycode, *max-keycode* — Type **card8**.

Returns *min-keycode* and *max-keycode* as multiple values. See the **display-max-keycode** and **display-min-keycode** functions for additional information.

display — A **display** object.

display-max-keycode *display* Function

Returns:
max-keycode — Type **card8**.

Returns the maximum keycode value for the specified *display.* This value is never greater than 255. Not all keycodes in the allowed range are required to have corresponding keys.

display — A **display** object.

display-max-request-length *display* Function

Returns:
max-request-length — Type **card16**.

Returns the maximum length of a request, in four-byte units, that is accepted by the specified *display.* Requests larger than this generate a length error, and the server will read and simply discard the entire request. This length is always at least 4096 (that is, requests of length up to and including 16384 bytes are accepted by all servers).

display — A **display** object.

display-min-keycode *display* Function

Returns:
min-keycode — Type **card8**.

Returns the minimum keycode value for the specified *display.* This value is never less than eight. Not all keycodes in the allowed range are required to have corresponding keys.

display — A **display** object.

display-motion-buffer-size *display* Function

Returns:
motion-buffer-size — Type **card32**.

Returns the approximate size of the motion buffer for the specified *display.* The server can retain the recent history of pointer motion at a finer granularity than is reported by **:motion-notify** events. Such history is available through the **motion-events** function.

display — A **display** object.

display-p *display* Function

Returns:
display-p — Type **boolean**.

Returns non-**nil** if *display* is a **display** object; otherwise, returns **nil**.

display-pixmap-formats *display* Function

Returns:
pixmap-formats — Type **list**.

Returns the list of **pixmap-format** values for the given *display.* This list contains one entry for each depth value. The entry describes the format used to represent images of that depth. An entry for a depth is included if any screen supports that depth, and all screens supporting that depth must support (only) the format for that depth.

display — A **display** object.

display-plist *display* Function

Returns:
plist — Type **list**.

Returns and (with **setf**) sets the property list for the specified *display.* This function provides a hook where extensions can add data.

display — A **display** object.

display-protocol-major-version *display* Function

Returns:
protocol-major-version — Type **card16**.

Returns the major version number of the X protocol associated with the specified *display.* In general, the major version would increment for incompatible changes. The returned protocol version number indicates the protocol the server actually supports. This might not equal the version supported by the client. The server can (but need not) refuse connections from clients that offer a different version than the server supports. A server can (but need not) support more than one version simultaneously.

display — A **display** object.

display-protocol-minor-version *display* Function

Returns:
protocol-minor-version — Type **card16**.

Returns the minor protocol revision number associated with the specified *display.* In general, the minor version would increment for small upward compatible changes in the X protocol.

display — A **display** object.

display-protocol-version *display* Function

Returns:
protocol-major-version, *protocol-minor-version* — Type **card16**.

Returns *protocol-major-version* and *protocol-minor-version* as multiple values. See the **display-protocol-major-version** and **display-protocol-minor-version** functions for additional information.

display — A **display** object.

display-resource-id-base *display* Function

Returns:
resource-id-base — Type **resource-id**.

Returns the *resource-id-base* value that was returned from the server during connection setup for the specified *display.* This is used in combination with the *resource-id-mask* to construct valid IDs for this connection.

display — A **display** object.

display-resource-id-mask *display* Function

Returns:
resource-id-mask — Type **resource-id**.

Returns the *resource-id-mask* that was returned from the server during connection setup for the specified *display*. The *resource-id-mask* contains a single contiguous set of bits (at least 18) which the client uses to allocate resource IDs for types **window**, **pixmap**, **cursor**, **font**, **gcontext**, and **colormap** by choosing a value with (only) some subset of these bits set, and **or**ing it with the *resource-id-base*. Only values constructed in this way can be used to name newly created server resources over this connection. Server resource IDs never have the top three bits set. The client is not restricted to linear or contiguous allocation of server resource IDs. Once an ID has been freed, it can be reused, but this should not be necessary.

An ID must be unique with respect to the IDs of all other server resources, not just other server resources of the same type. However, note that the value spaces of server resource identifiers, atoms, visualids, and keysyms are distinguished by context, and as such are not required to be disjoint (for example, a given numeric value might be both a valid window ID, a valid atom, and a valid keysym.)

display — A **display** object.

display-roots *display* Function

Returns:
roots — A list of screens.

Returns a list of all the **screen** structures available for the given *display*.

display — A **display** object.

display-vendor *display* Function

Returns:
vendor-name, *release-number* — Type **card32**.

Returns *vendor-name* and *release-number* as multiple values. See the **display-vendor-name** and **display-release-number** functions for additional information.

display — A **display** object.

display-vendor-name *display* Function

Returns:
vendor-name — Type **string**.

Returns a string that provides some vendor identification of the X server implementation associated with the specified *display*.

display — A **display** object.

display-version-number *display* Function

Returns:
version-number — Type **card16**.

Returns the X protocol version number for this implementation of CLX.

display — A **display** object.

display-xid *display* Function

Returns:
resource-allocator — Type **function**.

Returns the function that is used to allocate server resource IDs for this *display*.

display — A **display** object.

with-display *display* &body *body* Macro

This macro is for use in a multi-process environment. **with-display** provides exclusive access to the local **display** object for multiple request generation. It need not provide immediate exclusive access for replies. That is, if another process is waiting for a reply (while not in a **with-display**), then synchronization need not (but can) occur immediately. Except where noted, all routines effectively contain an implicit **with-display** where needed, so that correct synchronization is always provided at the interface level on a per-call basis. Nested uses of this macro work correctly. This macro does not prevent concurrent event processing (see **with-event-queue**).

display — A **display**.

Managing the Output Buffer

2.4 Most CLX functions cause output requests to be generated to an X server. Output requests are not transmitted immediately but instead are stored in an *output buffer* for the appropriate display. Requests in the output buffer are typically sent only when the buffer is filled. Alternatively, buffered requests can be sent prior to processing an event in the input event queue (see paragraph 12.3, Processing Events). In either case, CLX sends the output buffer automatically without explicit instructions from the client application.

However, in some cases, explicit control over the output buffer is needed, typically to ensure that the X server is in a consistent state before proceeding further. The **display-force-output** and **display-finish-output** functions allow a client program to synchronize with buffered output requests.

display-after-function *display* Function

Returns:
after-function — Type **function** or **null**.

Returns and (with **setf**) sets the *after-function* for the given *display*. If *after-function* is non-**nil**, it is a function that is called after every protocol request is generated, even those inside an explicit **with-display**, but never called from inside the *after-function*. The function is called inside the effective **with-display** for the associated request. The default value is **nil**. This can be set, for example, to #'**display-force-output** or #'**display-finish-outpu**t.

display — A **display** object.

display-force-output *display* Function

Forces any buffered output to be sent to the X server.

display — A **display** object.

display-finish-output *display* Function

Forces any buffered output to be sent to the X server and then waits until all requests have been received and processed. Any errors generated are read and handled by the display error handler. Any events generated by output requests are read and stored in the event queue.

display — A **display** object.

Closing the Display

2.5 To close or disconnect a display from the X server, use **close-display**.

close-display *display* Function

Closes the connection to the X server for the specified *display*. It destroys all server resources (**window**, **font**, **pixmap**, **colormap**, **cursor**, and **gcontext**), that the client application has created on this display, unless the close down mode of the server resource has been changed (see **set-close-down-mode**). Therefore, these server resources should never be referenced again. In addition, this function discards any output requests that have been buffered but have not yet been sent.

display — A **display** object.

SCREENS

Screens and Visuals

3.1 An X display supports graphical output to one or more *screens*. Each screen has its own root window and window hierarchy. Each window belongs to exactly one screen and cannot simultaneously appear on another screen.

The kinds of graphics hardware used by X screens can vary greatly in their support for color and in their methods for accessing raster memory. X uses the concept of a *visual type* (usually referred to simply as a *visual*) which uniquely identifies the hardware capabilities of a display screen. Fundamentally, a visual is represented by a **card29** integer ID, which uniquely identifies the visual type relative to a single display. CLX also represents a visual with a **visual-info** structure that contains other attributes associated with a visual (see paragraph 1.6, Data Types). A screen can support more than one depth (that is, pixel size), and for each supported depth, a screen may support more than one visual. However, it is more typical for a screen to have only a single depth and a single visual type.

A visual represents various aspects of the screen hardware, as follows:

- A screen can be color or gray-scale.
- A screen can have a colormap that is either writable or read-only.
- A screen can have a single colormap or separate colormaps for each of the red, green, and blue components. With separate colormaps, a pixel value is decomposed into three parts to determine indexes into each of the red, green, and blue colormaps.

CLX supports the following classes of visual types: **:direct-color**, **:gray-scale**, **:pseudo-color**, **:static-color**, **:static-gray**, and **:true-color**. The following tables show how the characteristics of a screen determine the class of its visual type.

For screens with a single colormap:

	Color	Gray-Scale
Read-only	**:static-color**	**:static-gray**
Writable	**:pseudo-color**	**:gray-scale**

For screens with red, green, and blue colormaps:

Read-only	**:true-color**	
Writable	**:direct-color**	**:gray-scale**

The visual class also indicates how screen colormaps are handled. See paragraph 9.1, Colormaps and Colors.

Screen Attributes

3.2 In CLX, each display screen is represented by a **screen** structure. The **display-roots** function returns the list of **screen** structures for the display. The following paragraphs discuss the attributes of CLX **screen** structures.

screen-backing-stores *screen* Function

Returns:
backing-stores-type — One of **:always**, **:never**, or **:when-mapped**.

Returns a value indicating when the *screen* supports backing stores, although it may be storage limited in the number of windows it can support at once. The value returned can be one of **:always**, **:never**, or **:when-mapped**.

screen — A **screen**.

screen-black-pixel *screen* Function

Returns:
black-pixel — Type **pixel**.

Returns the black pixel value for the specified *screen*.

screen — A **screen**.

screen-default-colormap *screen* Function

Returns:
default-colormap — Type **colormap**.

Returns the *default-colormap* for the specified *screen*. The *default-colormap* is initially associated with the root window. Clients with minimal color requirements creating windows of the same depth as the root may want to allocate from this map by default. Most routine allocations of color should be made out of this colormap.

screen — A **screen**.

screen-depths *screen* Function

Returns:
depths — Type **alist**.

Returns an association list that specifies what drawable depths are supported on the specified *screen*. Elements of the returned association list have the form (depth *visual**), where each *visual* is a **visual-info** structure. Pixmaps are supported for each depth listed, and windows of that depth are supported if at least one visual type is listed for the depth. A pixmap depth of one is always supported and listed, but windows of depth one might not be supported. A depth of zero is never listed, but zero-depth **:input-only** windows are always supported.

screen — A **screen**.

screen-event-mask-at-open *screen* Function

Returns:
event-mask-at-open — Type **mask32**.

Returns the initial root event mask for the specified *screen*.

screen — A **screen**.

screen-height *screen* Function

Returns:
height — Type **card16**.

Returns the *height* of the specified *screen* in pixel units.

screen — A **screen**.

screen-height-in-millimeters *screen* Function

Returns:
height-in-millimeters — Type **card16**.

Returns the height of the specified *screen* in millimeters. The returned height can be used with the width in millimeters to determine the physical size and the aspect ratio of the screen.

screen — A **screen**.

screen-max-installed-maps *screen* Function

Returns:
max-installed-colormaps — Type **card16**.

Returns the maximum number of colormaps that can be installed simultaneously with **install-colormap**.

screen — A **screen**.

screen-min-installed-maps *screen* Function

Returns:
min-installed-colormaps — Type **card16**.

Returns the minimum number of colormaps that can be guaranteed to be installed simultaneously.

screen — A **screen**.

screen-p *screen*Function

Returns:
screen-p — Type **boolean**.

Returns non-**nil** if the *screen* argument is a **screen** structure; otherwise, returns **nil**.

screen-plist *screen* Function

Returns:
plist — Type **list**.

Returns and (with **setf**) sets the property list for the specified *screen*. This function provides a hook where extensions can add data.

screen — A **screen**.

screen-root *screen* Function

Returns:
root-window — Type **window** or **null**.

Returns the *root-window* for the specified *screen*. This function is useful with functions that take a parent window as an argument. The class of the root window is always **:input-output**.

screen — A **screen**.

screen-root-depth *screen* Function

Returns:
root-window-depth — Type **image-depth**.

Returns the depth of the root window for the specified *screen*. Other depths can also be supported on this *screen*.

screen — A **screen**.

screen-root-visual *screen* Function

Returns:
root-window-visual — Type **card29**.

Returns the default visual type for the root window for the specified *screen*.

screen — A **screen**.

screen-save-unders-p *screen* Function

Returns:
save-unders-p — Type **boolean**.

If true, the server can support the save-under mode in **create-window** and in changing window attributes.

screen — A screen.

screen-white-pixel *screen* Function

Returns:
white-pixel — Type **pixel**.

Returns the white pixel value for the specified *screen*.

screen — A screen.

screen-width *screen* Function

Returns:
width — Type **card16**.

Returns the width of the specified *screen* in pixel units.

screen — A screen.

screen-width-in-millimeters *screen* Function

Returns:
width-in-millimeters — Type **card16**.

Returns the width of the specified *screen* in millimeters. The returned width can be used with the height in millimeters to determine the physical size and the aspect ratio of the screen.

screen — A screen.

4 WINDOWS AND PIXMAPS

Drawables

4.1 Both windows and pixmaps can be used as sources and destinations in graphics operations. These are collectively known as *drawables*. The following functions apply to both windows and pixmaps.

drawable-display *drawable* — Function

Returns the display for the specified *drawable*.

drawable — A **drawable** object.

drawable-equal *drawable-1 drawable-2* — Function

Returns true if the two arguments refer to the same server resource, and **nil** if they do not.

drawable-1, *drawable-2* — **drawable** objects.

drawable-id *drawable* — Function

Returns:
id — Type **resource-id**.

Returns the unique resource ID assigned to the specified *drawable*.

drawable — A **drawable** object.

drawable-p *drawable* — Function

Returns:
boole — Type **boolean**.

Returns true if the argument is a **drawable** and **nil** otherwise.

drawable-plist *drawable* — Function

Returns:
plist — A property list.

Returns and (with **setf**) sets the property list for the specified *drawable*. This function provides a hook where extensions can add data.

Creating Windows

4.2 A window is a **drawable** that can also receive input events. CLX represents a window with a **window** object. The **create-window** function creates a new **window** object.

create-window &key **:parent :x :y :width :height** (**:depth** 0) (**:border-width** 0) (**:class :copy**) (**:visual :copy**) **:background** **:border :gravity :bit-gravity :backing-store :backing-planes :backing-pixel** **:save-under :event-mask :do-not-propagate-mask :override-redirect :colormap** **:cursor** — Function

Returns:
window — Type **window**.

Creates and returns a window. A **:parent** window must be specified; the first window created by a client will have a root window as its **:parent**. The new window is initially unmapped and is placed on top of its siblings in the stacking order. A **:create-notify** event is generated by the server.

The **:class** of a window can be **:input-output** or **:input-only**. Windows of class **:input-only** cannot be used as the destination drawable for graphics output and can never receive **:exposure** events, but otherwise operate the same as **:input-output** windows. The **:class** can also be **:copy**, in which case the new window has the same class as its **:parent**.

For an **:input-output** window, the **:visual** and **:depth** must be a combination supported by the **:parent**'s screen, but the **:depth** need not be the same as the **:parent**'s. The **:parent** of an **:input-output** window must also be **:input-output**. A **:depth** of 0 means that the depth of the **:parent** is used.

For an **:input-only** window, the **:depth** must be zero, and the **:visual** must be supported by the **:parent**'s screen. The **:parent** of an **:input-only** window can be of any class. The only attributes that can be given for an **:input-only** window are **:cursor**, **:do-not-propagate-mask**, **:event-mask**, **:gravity**, and **:override-redirect**.

:parent — The parent window. This argument is required.

:x, **:y — int16** coordinates for the outside upper-left corner of the new window with respect to the origin (inside upper-left corner) of the **:parent**. These arguments are required.

:width, **:height — card16** values for the size of the new window. These arguments are required.

:depth — A **card16** specifying the depth of the new window.

:class — One of **:input-outpu**t, **:input-only**, or **:copy**.

:visual — A **card29** ID specifying the visual type of the new window.

:background, **:backing-pixel**, **:backing-planes**, **:backing-store**, **:bit-gravity**, **:border**, **:border-width**, **:colormap**, **:cursor**, **:do-not-propagate-mask**, **:event-mask**, **:gravity**, **:override-redirect**, **:save-under —** Initial attribute values for the new window. If **nil**, the default value is defined by the X protocol. See paragraph 4.3, Window Attributes.

Window Attributes

4.3 The following paragraphs describe the CLX functions used to return or change window attributes. Using the **with-state** macro improves the performance of attribute access by batching related accesses in the minimum number of server requests.

drawable-border-width *drawable* Function

Returns:
border-width — Type **card16**.

Returns the *border-width* of the *drawable* in pixels. It always returns zero if the *drawable* is a pixmap or an **:input-only** window. Used with **setf**, this function also changes the border width of the **:input-only** window. The default border width of a new window is zero.

Changing just the border width leaves the outer left corner of a window in a fixed position but moves the absolute position of the window's origin. It is an error to make the border width of an **:input-only** window nonzero.

When changing the border-width of a window, if the override-redirect attribute of the window is **:off** and some other client has selected **:substructure-redirect** on the parent, a **:configure-request** event is generated, and no further processing is performed. Otherwise, the border-width is changed.

drawable — A **drawable** object.

drawable-depth *drawable* Function

Returns:
depth — Type **card8**.

Returns the depth of the specified *drawable* (bits per pixel).

drawable — A **drawable** object.

drawable-height *drawable* Function

Returns:
inside-height — Type **card16**.

drawable-width *drawable* Function

Returns:
inside-width — Type **card16**.

These functions return the height or width of the *drawable*. These coordinates define the inside size of the *drawable*, in pixels. Used with **setf**, these functions also change the inside height or width of a window. However, the height or width of a pixmap cannot be changed.

Changing the width and height resizes a window without changing its position or stacking priority.

Changing the size of a mapped window may cause the window to lose its contents and generate an **:exposure** event. If a mapped window is made smaller, **:exposure** events are generated on windows that it formerly obscured.

When changing the size of a window, if the override-redirect attribute of the window is **:off** and some other client has selected **:substructure-redirect** on the parent, a **:configure-request** event is generated, and no further processing is performed. Otherwise, if another client has selected **:resize-redirect** on the window, a **:resize-request** event is generated, and the current inside width and height are maintained. Note that the override-redirect attribute of the window has no effect on **:resize-redirect** and that **:substructure-redirect** on the parent has precedence over **:resize-redirect** on the window.

When the inside size of the window is changed, the children of the window can move according to their window gravity. Depending on the window's bit gravity, the contents of the window can also be moved.

drawable — A **drawable** object.

drawable-x *drawable* Function

Returns:
outside-left — Type **int16**.

drawable-y *drawable* Function

Returns:
outside-top — Type **int16**.

These functions return the x or y coordinate of the specified *drawable*. They always return zero if the *drawable* is a pixmap. These coordinates define the location of the top left pixel of the window's border or of the window, if it has no border. Used with **setf**, these functions also change the x or y coordinate of a window. However, the x or y coordinate of a pixmap cannot be changed.

Changing the x and y coordinates moves a window without changing its size or stacking priority. Moving a mapped window generates **:exposure** events on any formerly obscured windows.

When changing the position of a window, if the override-redirect attribute of the window is **:off** and some other client has selected **:substructure-redirect** on the parent, a **:configure-request** event is generated, and no further processing is performed. Otherwise, the window is moved.

drawable — A **drawable** object.

window-all-event-masks *window* Function

Returns:
all-event-masks — Type **mask32**.

Returns the inclusive-or of the event masks selected on the specified *window* by all clients.

window — A **window**.

setf (**window-background**) *window background* Function

Returns:
background — Either a **pixel**, a **pixmap**, **:none**, or **:parent-relative**.

Changes the *background* attribute of the *window* to the specified value. This operation is not allowed on an **:input-only** window. Changing the background does not cause the window contents to be changed. Note that the background of a window cannot be returned from the X server. The default background of a new window is **:none**.

In general, the server automatically fills in exposed areas of the window when they are first made visible. A background pixmap is tiled to fill each area. However, if the background is **:none**, the server will not modify exposed areas. If the background is **:parent-relative**, the window and its parent must have the same depth. In this case, the window shares the same background as its parent. The parent's background is not copied and is reexamined whenever the window's background is required. If the background is **:parent-relative**, the background pixmap tile origin is the same as the parent's; otherwise, the tile origin is the window origin.

window — A **window**.

background — Either a **pixel**, a **pixmap**, **:none**, or **:parent-relative**.

window-backing-pixel *window* Function

Returns:
backing-pixel — Type **pixel**.

Returns and (with **setf**) changes the value of the backing-pixel attribute for the specified *window.* Changing the backing-pixel attribute of a mapped window may have no immediate effect. The default backing-pixel of a new window is zero.

window — A **window**.

window-backing-planes *window* Function

Returns:
backing-planes — Type **pixel**.

Returns and (with **setf**) changes the value of the backing-planes attribute for the specified *window.* Changing the backing-planes attribute of a mapped window may have no immediate effect. The default backing-planes of a new window is all one's.

window — A **window**.

window-backing-store *window* Function

Returns:
backing-store-type — One of **:always**, **:not-useful**, or **:when-mapped**.

Returns and (with **setf**) changes the value of the backing-store attribute for the specified *window*. Changing the backing-store attribute of an obscured window to **:when-mapped** or **:always** may have no immediate effect. The default backing-store of a new window is **:not-useful**.

window — A **window**.

window-bit-gravity *window* Function

Returns:
bit-gravity — Type **bit-gravity**.

Returns and (with **setf**) changes the bit-gravity attribute of the *window.* If a window is reconfigured without changing its inside width or height, the contents of the window move with the window and are not lost. Otherwise, the contents of the resized window are either moved or lost, depending on its bit-gravity attribute. The default bit-gravity of a new window is **:forget**.

For example, suppose a window's size is changed by *W* pixels in width and *H* pixels in height. The following table shows, for each bit-gravity value, the change in position (relative to the window origin) that results for each pixel of the window contents.

Bit-Gravity	**X Change**	**Y Change**
:center	*W/2*	*H/2*
:east	*W*	*H/2*
:north	*W/2*	0
:north-east	*W*	0
:north-west	0	0
:south	*W/2*	*H*
:south-east	W	H
:south-west	0	H
:west	0	H/2

A **:static** bit-gravity indicates the contents or window should not move relative to the origin of the root window.

A server can choose to ignore the specified bit-gravity attribute and use **:forget** instead. A **:forget** bit-gravity attribute indicates that the window contents are always discarded after a size change, even if backing-store or save-under attributes are **:on**. The window's background is displayed (unless it is **:none**), and zero or more **:exposure** events are generated.

window — A **window**.

setf (**window-border**) *window border* Function

Returns:
border — Either a **pixel**, a **pixmap**, or **:copy**.

Changes the *border* attribute of the *window* to the specified value. This operation is not allowed on an **:input-only** window. Changing the border attribute also causes the window border to be repainted. Note that the border of a window cannot be returned from the X server. The default border of a new window is **:copy**.

A border pixmap is tiled to fill the border. The border pixmap tile origin is the same as the background tile origin. A border pixmap and the window must have the same root and depth. If the border is **:copy**, the parent's border is copied and used; subsequent changes to the parent's border do not affect the window border.

window — A **window**.

border — Either a **pixel**, a **pixmap**, or **:copy**.

window-class *window* Function

Returns:
class — Either **:input-output** or **:input-only**.

Returns the *class* of the specified *window.*

window — A **window**.

window-colormap *window* Function

Returns:
colormap — Type **colormap** or **null**.

Returns and (with **setf**) changes the value of the colormap attribute for the specified *window.* A value of **:copy** is never returned, since the parent's colormap attribute is actually copied, but the attribute can be set to **:copy** in a **setf** form. Changing the colormap of a window (defining a new map, not changing the contents of the existing map) generates a **:colormap-notify** event. Changing the colormap of a visible window may have no immediate effect on the screen (see **install-colormap**). The default colormap of a new window is **:copy**.

window — A **window**.

window-colormap-installed-p *window* Function

Returns:
colormap-installed-p — Type **boolean**.

Returns non-**nil** if the colormap associated with this *window* is installed. Otherwise, this function returns **nil**.

window — A **window**.

setf (**window-cursor**) *window cursor* Function

Returns:
cursor — Type **cursor** or **:none**.

Changes the *cursor* attribute of the *window* to the specified value. Changing the cursor of a root window to **:none** restores the default cursor. Note that the cursor of window cannot be returned from the X server. The default cursor of a new window is **:none**.

window — A **window**.

cursor — Either **cursor** or **:none**.

window-display *window* Function

Returns:
display — Type **display**.

Returns the **display** object associated with the specified *window*.

window — A **window**.

window-do-not-propagate-mask *window* Function

Returns:
do-not-propagate-mask — Type **mask32**.

Returns and (with **setf**) changes the do-not-propagate-mask attribute for the window. The default do-not-propagate-mask of a new window is zero.

If a window receives an event from one of the user input devices, and if no client has selected to receive the event, the event can instead be propagated up the window hierarchy to the first ancestor for which some client has selected it. However, any event type selected by the do-not-propagate-mask is not be propagated. The types of events that can be selected by the do-not-propagate-mask are those of type **device-event-mask-class**. See paragraph 12.2, Selecting Events.

window — A **window**.

window-equal *window-1 window-2* Function

Returns:
boolean.

Returns non-**nil** if the two arguments are the same window, and **nil** if they are not.

window-1, *window-2* — The windows to compare for equality.

window-event-mask *window* Function

Returns:
event-mask — Type **mask32**.

Returns and (with **setf**) changes the value of the event-mask attribute for the *window*. The default event-mask of a new window is zero.

window — A **window**.

window-gravity *window* Function

Returns:
gravity — Type **win-gravity**.

Returns and (with **setf**) changes the gravity attribute of the *window*. If a parent window is reconfigured without changing its inside width or height, then all child windows move with the parent and are not changed. Otherwise, each child of the resized parent is moved, depending on the child's gravity attribute. The default gravity of a new window is **:north-west**.

For example, suppose the size of the window's parent is changed by *W* pixels in width and *H* pixels in height. The following table shows, for each possible gravity value, the resulting change in the window's position relative to its parent's origin. When the window is moved, two events are generated—a **:configure-notify** event followed by a **:gravity-notify** event.

Gravity	X Change	Y Change
:center	*W/2*	*H/2*
:east	*W*	*H/2*
:north	*W/2*	0
:north-east	*W*	0
:north-west	0	0
:south	*W/2*	*H*
:south-east	W	H
:south-west	0	H
:west	0	H/2

A **:static** gravity indicates that the position of the window should not move relative to the origin of the root window.

An **:unmap** gravity is like **:north-west**, except the window is also unmapped and an **:unmap-notify** event is generated. This **:unmap-notify** event is generated after the **:configure-notify** event is generated for the parent.

window — A **window**.

window-id *window* Function

Returns:
The **resource-id** of the window.

Returns the unique ID assigned to *window.*

window — A **window**.

window-map-state *window* Function

Returns:
map-state — One of **:unmapped**, **:unviewable**, or **:viewable**.

Returns the map state of *window.* A window is **:unviewable** if it is mapped but some ancestor is unmapped.

window — A **window**.

window-override-redirect *window* Function

Returns:
override-redirect — Either **:on** or **:off**.

Returns and (with **setf**) changes the value of the override-redirect attribute for *window.* The default override-redirect of a new window is **:off**.

The override-redirect attribute determines whether or not attempts to change window geometry or parent hierarchy can be *redirected* by a window manager or some other client. The functions that might be affected by the override-redirect attribute are **circulate-window-down**, **circulate-window-up**, **drawable-border-width**, **drawable-height**, **drawable-width**, **drawable-x**, **drawable-y**, **map-window**, and **window-priority**.

window — A **window**.

window-p *object* Function

Returns:
window-p — Type **boolean**.

Returns non-**nil** if the *object* argument is a window; otherwise, it returns **nil**.

window-plist *window* Function

Returns:
plist — A property list.

Returns and (with **setf**) sets the property list for the specified *window*. This function provides a hook where extensions can hang data.

window — A **window**.

setf (**window-priority** *window*) (&optional *sibling*) *mode* Function

Returns:
mode — One of **:above**, **:below**, **:bottom-if**, **:opposite**, or **:top-if**.

Changes the stacking priority element of the *window* to the specified value. It is an error if the *sibling* argument is specified and is not actually a sibling of the window. Note that the priority of an existing window cannot be returned from the X server.

When changing the priority of a window, if the override-redirect attribute of the window is **:off** and some other client has selected :substructure-redirect on the parent, a :configure-request event is generated, and no further processing is performed. Otherwise, the priority is changed.

window — A **window**.

sibling — An optional argument specifying that *window* is to be restacked relative to this sibling **window**.

mode — One of **:above**, **:below**, **:bottom-if**, **:opposite**, or **:top-if**.

window-save-under *window* Function

Returns:
save-under — Either **:on** or **:off**.

Returns and (with **setf**) changes the value of the save-under attribute for the specified *window*. Changing the save-under attribute of a mapped window may have no immediate effect.

window — A **window**.

window-visual *window* Function

Returns:
visual-type — Type **card29**.

Returns the *visual-type* associated with the specified *window*.

window — A **window**.

with-state *drawable* &body *body* Macro

Batches successive read and write accesses to window attributes and drawable geometry, in order to minimize the number of requests sent to the server. Batching occurs automatically within the dynamic extent of the *body*. The *body* is not executed within a **with-display** form.

All window attributes can be returned or changed in a single request. Similarly, all drawable geometry values can be returned or changed in a single request. **with-state** combines accesses to these values into the minimum number of server requests necessary to guarantee that each read access returns the current server state of the *drawable*. The number of server requests sent depends on the sequence of calls to reader and **setf** functions within the dynamic extent of the *body*. There are two groups of reader and **setf** functions—the Window Attributes group and the Drawable Geometry group—as shown in Table 4-1.

Table 4-1 **Groups of Reader and Setf Functions**

Group	Reader Functions	Setf Functions
Window Attributes	**window-all-event-masks** **window-backing-pixel** **window-backing-planes** **window-backing-store** **window-bit-gravity** **window-class** **window-colormap** **window-colormap-installed-p** **window-do-not-propagate-mask** **window-event-mask** **window-gravity** **window-map-state** **window-override-redirect** **window-save-under** **window-visual**	**window-background** **window-backing-pixel** **window-backing-planes** **window-backing-store** **window-bit-gravity** **window-border** **window-colormap** **window-cursor** **window-do-not-propagate-mask** **window-event-mask** **window-gravity** **window-override-redirect** **window-save-under**
Drawable Geometry	**drawable-border-width** **drawable-depth** **drawable-height** **drawable-root** **drawable-width** **drawable-x** **drawable-y**	**drawable-border-width** **drawable-height** **drawable-width** **drawable-x** **drawable-y** **window-priority**

The results from a sequence of calls to **setf** functions in a given group are cached and sent in a single server request, either upon exit from the *body* or when a reader function from the corresponding group is called.

with-state sends a single request to update all its cached values for the *drawable* before the first call to a reader function within the *body* and also before the first call to a reader function following a sequence of calls to **setf** functions from the corresponding group.

drawable — A **display**.

body — The forms in which attributes accesses are batched.

Stacking Order

4.4 Sibling windows can *stack* on top of each other. Windows above can *obscure* or *occlude* lower windows. This relationship between sibling windows is known as the stacking order. The **window-priority** function can be used to change the stacking order of a single window. CLX also provides functions to raise or lower children of a window. Raising a mapped window can generate **:exposure** events for the window and any mapped subwindows that were formerly obscured. Lowering a mapped window can generate **:exposure** events on any windows it formerly obscured.

circulate-window-down *window* Function

Lowers the highest mapped child of the specified *window* that partially or completely occludes another child to the bottom of the stack. Completely unobscured children are unaffected. Exposure processing is performed on formerly obscured windows.

If some other client has selected **:substructure-redirect** on the *window*, a **:circulate-request** event is generated, and no further processing is performed. Otherwise, the child window is lowered and a **:circulate-notify** event is generated if the *window* is actually restacked.

window — A **window**.

circulate-window-up *window* Function

Raises the lowest mapped child of the specified *window* that is partially or completely occluded by another child to the top of the stack. Completely unobscured children are unaffected. Exposure processing is performed on formerly obscured windows.

If another client has selected **:substructure-redirect** on the *window*, a **:circulate-request** event is generated, and no further processing is performed. Otherwise, the child window is raised and a **:circulate-notify** event is generated if the *window* is actually restacked.

window — A **window**.

Window Hierarchy

4.5 All the windows in X are arranged in a strict hierarchy. At the top of the hierarchy are the root windows, which cover the display screens. Each root window is partially or completely covered by its child windows. All windows, except for root windows, have parents. Child windows can have their own children. In this way, a tree of arbitrary depth on each screen can be created. CLX provides several functions for examining and modifying the window hierarchy.

drawable-root *drawable* Function

Returns:
root-window — Type **window**.

Returns the root window of the specified *drawable*.

drawable — A **drawable**.

query-tree *window* &key (**:result-type 'list**) Function

Returns:
children — Type **sequence** of **window**.
parent — Type **window** or **null**.
root — Type **window**.

Returns the *children* windows, the *parent* window, and the *root* window for the specified *window*. The children are returned as a sequence of windows in current stacking order, from bottom-most (first) to top-most (last). The **:result-type** specifies the type of children sequence returned.

window — A **window**.

:result-type — A valid type specifier for a sub-type of **sequence**. The default is a **list**.

reparent-window *window parent x y* Function

Changes a *window*'s *parent* within a single screen. There is no way to move a window between screens.

The specified *window* is reparented by inserting it as a child of the specified *parent*. If the *window* is mapped, an **unmap-window** operation is automatically performed on the specified *window*. The *window* is then removed from its current position in the hierarchy and inserted as the child of the specified *parent*. The *window* is placed on top in the stacking order with respect to sibling windows.

After reparenting the specified *window,* a **:reparent-notify** event is generated. The override-redirect attribute of the *window* is passed on in this event. Window manager clients normally should ignore this event if this attribute is **:on**. See Section 12, Events and Input, for more information on **:reparent-notify** event processing. Finally, if the specified *window* was originally mapped, a **map-window** operation is automatically performed on it.

The X server performs normal exposure processing on formerly obscured windows. It might not generate **:exposure** events for regions from the initial **unmap-window** operation if they are immediately obscured by the final **map-window** operation.

It is an error if any of the following are true:

- The new *parent* window is not on the same screen as the old parent window.
- The new *parent* window is the specified *window* or an inferior of the specified *window.*
- The specified *window* has a **:parent-relative** background attribute and the new *parent* window is not the same depth as the specified *window.*

window — A **window**.

parent — The new parent **window**.

x, *y* — The position (type **int16**) of the *window* in its new *parent*. These coordinates are relative to the *parent*'s origin, and specify the new position of the upper, left, outer corner of the *window.*

translate-coordinates *source source-x source-y destination* Function

Returns:
destination-x — Type **int16** or **null**.
destination-y — Type **int16** or **null**.
destination-child — Type **window** or **null**.

Returns the position defined by *source-x* and *source-y* (relative to the origin of the *source* window), expressed as coordinates relative to the origin of the *destination* window.

source — A **window** defining the source coordinate system.

source-x, *source-y* — Coordinates (**int16**) relative to the origin of the *source* **window**.

destination — A **window** defining the destination coordinate system.

Mapping Windows

4.6 A window is considered mapped if a **map-window** call has been made on it. When windows are first created, they are not mapped because an application may wish to create a window long before it is mapped to the screen. A mapped window may not be visible on the screen for one of the following reasons:

- It is obscured by another opaque sibling window.
- One of its ancestors is not mapped.
- It is entirely clipped by an ancestor.

A subwindow will appear on the screen as long as all of its ancestors are mapped and not obscured by a sibling or clipped by an ancestor. Mapping a window that has an unmapped ancestor does not display the window, but marks it as eligible for display when the ancestor becomes mapped. Such a window is called unviewable. When all its ancestors are mapped, the window becomes viewable and remains visible on the screen if not obscured by any sibling or ancestor.

Any output to a window not visible on the screen is discarded. **:exposure** events are generated for the window when part or all of it becomes visible on the screen. A client only receives the **:exposure** events if it has selected them. Mapping or unmapping a window does not change its stacking order priority.

map-window *window* Function

Maps the *window*. This function has no effect when the *window* is already mapped.

If the override-redirect attribute of the *window* is **:off** and another client has selected **:substructure-redirect** on the parent window, the X server generates a **:map-request** event and the **map-window** function does not map the *window*. Otherwise, the *window* is mapped, and the X server generates a **:map-notify** event.

If the *window* becomes visible and no earlier contents for it are remembered, **map-window** tiles the window with its background. If no background was defined for the window, the existing screen contents are not altered, and the X server generates one or more **:exposure** events. If a backing-store was maintained while the window was unmapped, no **:exposure** events are generated. If a backing-store will now be maintained, a full window exposure is always generated. Otherwise, only visible regions may be reported. Similar tiling and exposure take place for any newly viewable inferiors.

map-window generates **:exposure** events on each **:input-output** window that it causes to become visible.

window — A **window**.

map-subwindows *window* Function

Maps all child windows for a specified *window* in top-to-bottom stacking order. The X server generates an **:exposure** event on each newly visible window. This function is much more efficient than mapping each child individually.

window — A **window**.

unmap-window *window* Function

Unmaps the specified *window* and causes the X server to generate an **:unmap-notify** event. If the specified *window* is already unmapped, **unmap-window** has no effect. Normal exposure processing on formerly obscured windows is performed. Any child window is no longer viewable. Unmapping the *window* generates **:exposure** events on windows that were formerly obscured by *window* and its children.

window — A **window**.

unmap-subwindows *window* Function

Unmaps all child windows for the specified *window* in bottom to top stacking order. The X server generates an **:unmap-notify** event on each child and **:exposure** events on formerly obscured windows. Using this function is much more efficient than unmapping child windows individually.

window — A **window**.

Destroying Windows

4.7 CLX provides functions to destroy a window or destroy all children of a window. Note that by default, windows are destroyed when a connection is closed. For further information, see paragraph 2.4, Closing the Display, and paragraph 12.4, Client Termination.

destroy-window *window* Function

Destroys the specified *window* as well as all of its inferiors. The windows should never again be referenced. If the specified *window* is mapped, it is automatically unmapped. The window and all of its inferiors are then destroyed, and a **:destroy-notify** event is generated for each window. The ordering of the **:destroy-notify** events is such that for any given window being destroyed, **:destroy-notify** is generated on the window's inferiors before being generated on the window. The ordering among siblings and across sub-hierarchies is not otherwise constrained. If the *window* is a root window, no windows are destroyed. Destroying a mapped window generates **:exposure** events on other windows that the mapped window obscured.

window — A **window**.

destroy-subwindows *window* Function

Destroys all inferiors of the specified *window*, in bottom to top stacking order. The X server generates a **:destroy-notify** event for each window. This is much more efficient than deleting many windows individually. The inferiors should never be referenced again.

window — A **window**.

Pixmaps

4.8 A *pixmap* is a three-dimensional array of bits. A pixmap is normally thought of as a two-dimensional array of pixels, where each pixel can be a value from 0 to $2^n - 1$, where *n* is the depth of the pixmap. A pixmap can also be thought of as a stack of *n* bitmaps. A *bitmap* is a single bit pixmap of depth 1. CLX provides functions to:

- Create or free a pixmap
- Test if an object is a pixmap
- Test if two pixmap objects are equal
- Return the pixmap resource ID from a **pixmap** object

Note that pixmaps can only be used on the screen where they were created. Pixmaps are off-screen server resources that are used for a number of operations. These include defining patterns for cursors or as the source for certain raster operations.

create-pixmap &key **:width :height :depth :drawable** — Function

Returns:
pixmap — Type **pixmap**.

Creates a pixmap of the specified **:width**, **:height**, and **:depth**. It is valid to pass a window whose class is **:input-only** as the **:drawable** argument. The **:width** and **:height** arguments must be nonzero. The **:depth** must be supported by the screen of the specified **:drawable**.

:width, **:height** — The nonzero width and height (type **card16**).

:depth — The depth (type **card8**) of the pixmap.

:drawable — A **drawable** which determines the screen where the pixmap will be used.

free-pixmap *pixmap* — Function

Allows the X server to free the pixmap storage when no other server resources reference it. The pixmap should never be referenced again.

pixmap — A **pixmap**.

pixmap-display *pixmap* — Function

Returns:
display — Type **display**.

Returns the **display** object associated with the specified *pixmap*.

pixmap — A **pixmap**.

pixmap-equal *pixmap-1 pixmap-2* — Function

Returns true if the two arguments refer to the same server resource, and **nil** if they do not.

pixmap-1, *pixmap-2* — A three-dimensional array of bits to be tested.

pixmap-id *pixmap* — Function

Returns:
id — Type **resource-id**.

Returns the unique resource ID that has been assigned to the specified *pixmap*.

pixmap — A **pixmap**.

pixmap-p *object* Function

Returns:
pixmap — Type **boolean**.

Returns true if the argument is a **pixmap** object and **nil** otherwise.

pixmap-plist *pixmap* Function

Returns:
plist — A property list.

Returns and (with **setf**) sets the property list for the specified *pixmap*. This function provides a hook where extensions can add data.

pixmap — A **pixmap**.

GRAPHICS CONTEXTS

5

Introduction

5.1 Clients of the X Window System specify the visual attributes of graphical output primitives by using *graphics contexts*. A graphics context is a set of graphical attribute values such as foreground color, font, line style, and so forth. Like a window, a graphics context is another kind of X server resource which is created and maintained at the request of a client program. The client program, which may use several different graphics contexts at different times, is responsible for specifying a graphics context to use with each graphical output function.

CLX represents a graphics context by an object of type **gcontext** and defines functions to create, modify, and manipulate **gcontext** objects. By default, CLX also records the contents of graphics contexts in a cache associated with each display. This local caching of graphics contexts has two important advantages:

1. Communication efficiency — Changes to attribute values in a **gcontext** are first made only in the local cache. Just before a **gcontext** is actually used, CLX automatically sends any changes to the X server, batching all changes into a single request.

2. Inquiring **gcontext** contents — Accessor functions can be used to return the value of any individual **gcontext** component by reading the copy of the **gcontext** from the cache. This kind of inquiry is not supported by the basic X protocol. There is no way for a client program to request an X server to return the contents of a **gcontext**.

Caching graphics contexts can result in a synchronization problem if more than one client program modifies a graphics context. However, this problem is unusual. Sharing a graphics context among several clients, while possible, is not expected to be useful and is not very easy to do. At any rate, a client program can choose to not cache a **gcontext** when it is created.

Each client program must determine its own policy for creating and using graphics contexts. Depending on the display hardware and the server implementation, creating a new graphics context can be more or less expensive than modifying an existing one. In general, some amount of graphics context information can be cached in the display hardware, in which case modifying the hardware cache is faster than replacing it. Typical display hardware can cache only a small number of graphics contexts. Graphics output is fastest when only a few graphics contexts are used without heavy modifications.

This section explains the CLX functions used to:

- Create a graphics context
- Return the contents of a graphics context
- Change the contents of a graphics context
- Copy a graphics context
- Free a graphics context

Creating Graphics Contexts

5.2 To create a graphics context, use **create-gcontext.**

create-gcontext &key **:arc-mode :background** (**:cache-p** t) **:cap-style** Function
:clip-mask :clip-ordering :clip-x :clip-y :dash-offset :dashes
:drawable :exposures :fill-rule :fill-style :font :foreground
:function :join-style :line-style :line-width :plane-mask :stipple
:subwindow-mode :tile :ts-x :ts-y
Returns:
gcontext — Type **gcontext**.

Creates, initializes, and returns a graphics context (**gcontext**). The graphics context can only be used with destination drawables having the same root and depth as the specified **:drawable**. If **:cache-p** is non-**nil**, the graphics context state is cached locally, and changing a component has no effect unless the new value differs from the cached value. Changes to a graphics context (**setf** and **with-gcontext**) are always deferred regardless of the cache mode and sent to the server only when required by a local operation or by an explicit call to **force-gcontext-changes**.

:cache-p — Specifies if this graphics context should be cached locally by CLX. If **nil** then the state is not cached, otherwise a local cache is kept.

:drawable — The **drawable** whose root and depth are to be associated with this graphics context. This is a required keyword argument.

:arc-mode, **:background**, **:cap-style**, **:clip-mask**, **:clip-ordering**, **:clip-x**, **:clip-y**, **:dash-offset**, **:dashes**, **:exposures**, **:fill-rule**, **:fill-style**, **:font**, **:foreground**, **:function**, **:join-style**, **:line-style**, **:line-width**, **:plane-mask**, **:stipple**, **:subwindow-mode**, **:tile**, **:ts-x**, **:ts-y** — Initial attribute values for the graphics context.

All of the graphics context components are set to the values that are specified by the keyword arguments, except that a value of **nil** causes the default value to be used. These default values are as follows:

Component	Default Value
arc-mode	**:pie-slice**
background	1
cap-style	**:butt**
clip-mask	**:none**
clip-ordering	**:unsorted**
clip-x	0
clip-y	0
dash-offset	0
dashes	4 (that is, the list '(4, 4))
exposures	**:on**
fill-rule	**:even-odd**
fill-style	**:solid**
font	server dependent
foreground	0
function	**boole-1**
join-style	**:miter**
line-style	**:solid**
line-width	0
plane-mask	A bit mask of all ones
stipple	Pixmap of unspecified size filled with ones
subwindow-mode	**:clip-by-children**
tile	Pixmap of an unspecified size filled with the foreground pixel (that is, the client-specified pixel if any, or else 0)
ts-x	0
ts-y	0

Note that foreground and background do not default to any values that are likely to be useful on a color display. Since specifying a **nil** value means use the default, this implies for clip-mask that an empty rectangle sequence cannot be specified as an empty list; **:none** must be used instead. Specifying a **stringable** for font causes an implicit **open-font** call to occur.

Graphics Context Attributes

5.3 The following paragraphs describe the CLX functions used to return or change the attributes of a **gcontext**. Functions that return the contents of a **gcontext** return **nil** if the last value stored is unknown (for example, if the **gcontext** was not cached or if the **gcontext** was not created by the inquiring client).

gcontext-arc-mode *gcontext* Function

Returns:
arc-mode — Either **:chord** or **:pie-slice**.

Returns and (with **setf**) changes the arc-mode attribute of the specified graphics context.

The arc-mode attribute of a graphics context controls the kind of filling, if any, to be done by the **draw-arcs** function. A value of **:chord** specifies that arcs are filled inward to the chord between the end points of the arc. **:pie-slice** specifies that arcs are filled inward to the center point of the arc, creating a pie slice effect.

gcontext — A **gcontext**.

gcontext-background *gcontext* Function

Returns:
background — Type **card32**.

Returns and (with **setf**) changes the background attribute of the specified graphics context.

The background attribute specifies the pixel value drawn for pixels that are not set in a bitmap and for pixels that are cleared by a graphics operation, such as the gaps in dashed lines.

gcontext — A **gcontext**.

gcontext-cache-p *gcontext* Function

Returns:
cache-p — Type **boolean**.

Returns and (with **setf**) changes the local cache mode for the *gcontext*. If true, the state of the *gcontext* is cached by CLX and changes to its attributes have no effect unless the new value differs from its cached value.

gcontext — A **gcontext**.

gcontext-cap-style *gcontext* Function

Returns:
cap-style — One of **:butt**, **:not-last**, **:projecting**, or **:round**.

Returns and (with **setf**) changes the cap-style attribute of the specified graphics context.

The cap-style attribute of a graphics context defines how the end points of a path are drawn. The possible values and their interpretations are as follows:

Cap-Style	Interpretation
:butt	Square at the end point (perpendicular to the slope of the line) with no projection beyond.
:not-last	Equivalent to **:butt**, except that for a line-width of zero or one the final end point is not drawn.
:projecting	Square at the end, but the path continues beyond the end point for a distance equal to half the line-width. This is equivalent to **:butt** for line-width zero or one.
:round	A circular arc with the radius equal to 1/2 of the line-width, centered on the end point. This is equivalent to **:butt** for line-width zero or one.

The following table describes what happens when the end points of a line are identical. The effect depends on both the cap style and line width.

Cap-Style	Line-Width	Effect
:butt	thin	Device dependent, but the desired effect is that a single pixel is drawn.
:butt	wide	Nothing is drawn.
:not-last	thin	Device dependent, but the desired effect is that nothing is drawn.
:projecting	thin	Same as **:butt** with thin line-width.
:projecting	wide	The closed path is a square, aligned with the coordinate axes, centered at the end point, with sides equal to the line-width.
:round	wide	The closed path is a circle, centered at the end point, with diameter equal to the line-width.
:round	thin	Same as **:butt** with thin line-width.

gcontext — A **gcontext**.

gcontext-clip-mask *gcontext* &optional *ordering* Function

Returns and (with **setf**) changes the clip-mask attribute of the graphics context.

When changing the clip-mask attribute, the new clip-mask can be specified as a pixmap or a **rect-seq** or as the values **:none** or **nil**. The ordering argument can be specified only with **setf** when the new clip-mask is a **rect-seq**.

The clip-mask attribute of a graphics context affects all graphics operations and is used to restrict output to the destination drawable. The clip-mask does not clip the source of a graphics operation. A value of **:none** for clip-mask indicates that no clipping is to be done.

If a pixmap is specified as the clip-mask, it must have depth one and the same root as the specified graphics context. Pixels where the clip-mask has a one bit are drawn. Pixels outside the area covered by the clip-mask or where the clip-mask has a zero bit are not drawn.

If a sequence of rectangles is specified as the clip-mask, the output is clipped to remain contained within the rectangles. The rectangles should be non-intersecting, or the results of graphics operations will be undefined. The rectangle coordinates are interpreted relative to the clip origin. Note that the sequence of rectangles can be empty, which effectively disables output. This is the opposite of setting the clip-mask to **:none**.

If known by the client, the ordering of clip-mask rectangles can be specified to provide faster operation by the server. A value of **:unsorted** means the rectangles are in arbitrary order. A value of **:y-sorted** means that the rectangles are non-decreasing in their Y origin. A **:yx-sorted** value is like **:y-sorted** with the additional constraint that all rectangles with an equal Y origin are non-decreasing in their X origin. A **:yx-banded** value additionally constrains **:yx-sorted** by requiring that, for every possible Y scan line, all rectangles that include that scan line have an identical Y origins and Y extents. If incorrect ordering is specified, the X server may generate an error, but it is not required to do so. If no error is generated, the results of the graphics operations are undefined.

gcontext — A **gcontext**.

ordering — One of **:unsorted**, **:y-sorted**, **:yx-banded**, **:yx-sorted**, or **nil**.

gcontext-clip-x *gcontext* Function

Returns:
clip-x — Type **int16**.

Returns and (with **setf**) changes the clip-x attribute of the specified graphics context.

The clip-x and clip-y attributes specify the origin for the clip-mask, whether it is a pixmap or a sequence of rectangles. These coordinates are interpreted relative to the origin of whatever destination drawable is specified in a graphics operation.

gcontext — A **gcontext**.

gcontext-clip-y *gcontext* Function

Returns:
clip-y — Type **int16**.

Returns and (with **setf**) changes the clip-y attribute of the specified graphics context.

The clip-x and clip-y attributes specify the origin for the clip-mask, whether it is a pixmap or a sequence of rectangles. These coordinates are interpreted relative to the origin of whatever destination drawable is specified in a graphics operation.

gcontext — A **gcontext**.

gcontext-dash-offset *gcontext* Function

Returns:
dash-offset — Type **card16**.

Returns and (with **setf**) changes the dash-offset attribute of the specified graphics context.

The dash-offset attribute of a graphics context defines the phase of the pattern contained in the dashes attribute. This phase specifies how many elements (pixels) into the path the pattern should actually begin in any single graphics operation. Dashing is continuous through path elements combined with a join-style, but is reset to the dash-offset each time a cap-style is applied at a line end point.

gcontext — A **gcontext**.

gcontext-dashes *gcontext* Function

Returns:
dashes — Type **sequence** or **card8**.

Returns and (with **setf**) changes the dashes attribute of the specified graphics context. The sequence must be non-empty and the elements must be non-zero **card8** values.

The dashes attribute in a graphics context specifies the pattern that is used for graphics operations which use the dashed line styles. It is a non-**nil** sequence with each element representing the length of a single dash or space. The initial and alternating elements of the dashes are the even dashes, while the others are the odd dashes. An odd length sequence is equivalent to the same sequence concatenated with itself to produce an even length sequence. All of the elements of a dashes sequence must be non-zero.

Specifying a single integer value, *N*, for the dashes attribute is an abbreviated way of specifying a two element sequence with both elements equal to the specified value [*N*, *N*].

The unit of measure for dashes is the same as in the ordinary coordinate system. Ideally, a dash length is measured along the slope of the line, but server implementations are only required to match this ideal for horizontal and vertical lines.

gcontext — A **gcontext**.

gcontext-display *gcontext* Function

Returns:
display — Type **display**.

Returns the **display** object associated with the specified *gcontext*.

gcontext — A **gcontext**.

gcontext-equal *gcontext-1 gcontext-2* Function

Returns:
equal-p — Type **boolean**.

Returns true if the two arguments refer to the same server resource, and **nil** if they do not.

gcontext-1, *gcontext-2* — A **gcontext**.

gcontext-exposures *gcontext* Function

Returns:
exposures — Either **:off** or **:on**.

Returns and (with **setf**) changes the exposures attribute of the specified graphics context.

The exposures attribute in a graphics context controls the generation of **:graphics-exposure** events for calls to the **copy-area** and **copy-plane** functions. If **:on**, **:graphics-exposure** events will be reported when calling the **copy-area** and **copy-plane** functions with this graphics context. Otherwise, if **:off**, the events will not be reported.

gcontext — A **gcontext**.

gcontext-fill-rule *gcontext* Function

Returns:
fill-rule — Either **:even-odd** or **:winding**.

Returns and (with **setf**) changes the fill-rule attribute of the specified graphics context.

The fill-rule attribute in a graphics context specifies the rule used to determine the interior of a filled area. It can be specified as either **:even-odd** or **:winding**.

The **:even-odd** rule defines a point to be inside if any infinite ray starting at the point crosses the border an odd number of times. Tangencies do not count as a crossing.

The **:winding** rule defines a point to be inside if any infinite ray starting at the point crosses an unequal number of clockwise and counterclockwise directed border segments. A clockwise directed border segment crosses the ray from left to right as observed from the point. A counterclockwise segment crosses the ray from right to left as observed from the point. The case where a directed line segment is coincident with the ray is uninteresting because you can simply choose a different ray that is not coincident with a segment.

For both **:even-odd** and **:winding**, a point is infinitely small, and the border is an infinitely thin line. A pixel is inside if the center point of the pixel is inside, and the center point is not on the border. If the center point is on the border, the pixel is inside if, and only if, the polygon interior is immediately to its right (x increasing direction). Pixels with centers along a horizontal edge are a special case and are inside if, and only if, the polygon interior is immediately below (y increasing direction).

gcontext — A **gcontext**.

gcontext-fill-style *gcontext* Function

Returns:
fill-style — One of **:opaque-stippled**, **:solid**, **:stippled**, or :**tiled**.

Returns and (with **setf**) changes the fill-style attribute of the specified graphics context.

The fill-style attribute of a graphics context defines the contents of the source for line, text, and fill graphics operations. It determines whether the source image is drawn with a solid color, a tile, or a stippled tile. The possible values and their meanings are as follows:

Fill-Style	**Meaning**
:opaque-stippled	Filled with a tile with the same width and height as stipple, but with the background value used everywhere stipple has a zero and the foreground pixel value used everywhere stipple has a one.
:solid	Filled with the foreground pixel value.
:stippled	Filled with the foreground pixel value masked by stipple.
:tiled	Filled with tile.

When drawing lines with line-style **:double-dash**, the filling of the odd dashes are controlled by the fill-style in the following manner:

Fill-Style	**Effect**
:opaque-stippled	Same as for even dashes.
:solid	Filled with the background pixel value.
:stippled	Filled with the background pixel value masked by stipple.
:tiled	Filled the same as the even dashes.

gcontext — A **gcontext**.

gcontext-font *gcontext* &optional *metrics-p* Function

Returns:
font — Type **font** or **null**.

Returns and (with **setf**) changes the *font* attribute of the specified graphics context. If the stored font is known, it is returned. If it is not known and the *metrics-p* argument is **nil**, then **nil** is returned. If the font is not known and *metrics-p* is true, then a pseudo-font is constructed and returned. For a constructed pseudo-font, full metric and property information can be obtained, but it does not have a name or a resource ID, and attempts to use it where a resource ID is required results in an invalid-font error.

The font attribute in a graphics context defines the default text font used in text drawing operations. When setting the value of the font attribute, either a **font** object or a font name can be used. If a font name is passed, **open-font** is call automatically to get the **font** object.

gcontext — A **gcontext**.

metrics-p — Specifies whether a pseudo-font is returned when the real font stored in the graphics context is not known. The default is **nil**, which means do not return a pseudo-font.

gcontext-foreground *gcontext* Function

Returns:
foreground — Type **card32**.

Returns and (with **setf**) changes the foreground attribute of the specified graphics context.

The foreground attribute of a graphics context specifies the pixel value drawn for set bits in a bitmap and for bits set by a graphics operation.

gcontext — A **gcontext**.

gcontext-function *gcontext* Function

Returns:
function — Type **boole-constant**.

Returns the *function* of the specified graphics context.

In all graphic operations, given a source pixel and a corresponding destination pixel, the resulting pixel drawn is computed bitwise on the bits of the source and destination pixels. That is, a logical operation is used to combine each bit plane of corresponding source and destination pixels. The graphics context function attribute specifies the logical operation used via one of the 16 operation codes defined by Common Lisp for the **boole** function.

The following table shows each of the logical operation codes that can be given by the function attribute. For each operation code, its result is shown as a logical function of a source pixel *S* and a destination pixel *D*.

Symbol	Result
boole-1	*S*
boole-2	*D*
boole-andc1	(logandc1 *S D*)
boole-andc2	(logandc2 *S D*)
boole-and	(logand *S D*)
boole-c1	(lognot *S*)
boole-c2	(lognot *D*)
boole-clr	0
boole-eqv	(logeqv *S D*)
boole-ior	(logior *S D*)
boole-nand	(lognand *S D*)
boole-nor	(lognor *S D*)
boole-orc1	(logorc1 *S D*)
boole-orc2	(logorc2 *S D*)
boole-set	1
boole-xor	(logxor *S D*)

gcontext — A **gcontext**.

gcontext-id *gcontext* Function

Returns:
id — Type **resource-id**.

Returns the unique ID that has been assigned to the specified graphics context.

gcontext — A **gcontext**.

gcontext-join-style *gcontext* Function

Returns:
join-style — One of **:bevel**, **:miter**, or **:round**.

Returns and (with **setf**) changes the join-style attribute of the specified graphics context.

The join-style attribute of a graphics context defines how the segment intersections are drawn for wide polylines. The possible values and their interpretations are as follows:

Join-Style	**Interpretation**
:bevel	Uses **:butt** end point styles with the triangular notch filled.
:miter	The outer edges of two lines extend to meet at an angle.
:round	A circular arc with diameter equal to the line-width, centered on the join point.

When the end points of a polyline segment are identical, the effect is as if the segment was removed from the polyline. When a polyline is a single point, the effect is the same as when the cap-style is applied at both end points.

gcontext — A **gcontext**.

gcontext-line-style *gcontext* Function

Returns:
line-style — One of **:dash**, **:double-dash**, or **:solid**.

Returns and (with **setf**) changes the line-style attribute of the specified graphics context.

The line-style attribute of a graphics context specifies how (which sections of) lines are drawn for a path in graphics operations. The possible values and their meanings are as follows:

Line-Style	**Meaning**
:solid	The full path is drawn.
:double-dash	The full path is drawn, but the even dashes are filled differently than the odd dashes. The **:butt** style is used where even and odd dashes meet (see paragraph 5.4.7, Fill-Rule and Fill-Style).
:on-off-dash	Only the even dashes are drawn, with cap-style applied to all internal ends of the individual dashes, except **:not-last** is treated as **:butt**.

gcontext — A **gcontext**.

gcontext-line-width *gcontext* Function

Returns:
line-width — Type **card16**.

Returns the *line-width* of the specified graphics context.

The line-width is measured in pixels and can be greater than or equal to one (wide line) or can be the special value zero (thin line).

Wide lines are drawn centered on the path described by the graphics operation. Unless otherwise specified by the join-style or cap-style, the bounding box of a wide line with end points [x1, y1], [x2, y2], and width w is a rectangle with vertices at the following real coordinates:

[x1 – (w**sin*/2), y1 + (w**cos*/2)], [x1+ (w**sin*/2), y1 – (w**cos*/2)],
[x2 – (w**sin*/2), y2 + (w**cos*/2)], [x2 + (w**sin*/2), y2 – (w**cos*/2)]

where *sin* is the sine of the angle of the line and *cos* is the cosine of the angle of the line. A pixel is part of the line and, hence, is drawn if the center of the pixel is fully inside the bounding box (which is viewed as having infinitely thin edges). If the center of the pixel is exactly on the bounding box, it is part of the line if, and only if, the interior is immediately to its right (x increasing direction). Pixels with centers on a horizontal edge are a special case and are part of the line if, and only if, the interior is immediately below (y increasing direction).

Thin lines (zero line-width) are always one pixel wide lines drawn using an unspecified, device dependent algorithm. There are only two constraints on this algorithm.

1. If a line is drawn unclipped from [x1,y1] to [x2,y2] and if another line is drawn unclipped from [x1+dx,y1+dy] to [x2+dx,y2+dy], a point [x,y] is touched by drawing the first line if, and only if ,the point [x+dx,y+dy] is touched by drawing the second line.

2. The effective set of points comprising a line cannot be affected by clipping. That is, a point is touched in a clipped line if, and only if, the point lies inside the clipping region and the point would be touched by the line when drawn unclipped.

A wide line drawn from [x1,y1] to [x2,y2] always draws the same pixels as a wide line drawn from [x2,y2] to [x1,y1], not counting cap-style and join-style. Implementors are encouraged to make this property true for thin lines, but it is not required. A line-width of zero may differ from a line-width of one in which pixels are drawn. This permits the use of many manufacturer's line drawing hardware, which may run much faster than the more precisely specified wide lines.

In general, drawing a thin line is faster than drawing a wide line of width one. However, because of their different drawing algorithms, thin lines may not mix well, aesthetically speaking, with wide lines. If it is desirable to obtain precise and uniform results across all displays, a client should always use a line-width of one, rather than a line-width of zero.

gcontext — A **gcontext**.

gcontext-p *gcontext* Function

Returns:
gcontext — Type **boolean**.

Returns non-**nil** if the argument is a graphics context and **nil** otherwise.

gcontext-plane-mask *gcontext* Function

Returns:
plane-mask — Type **card32**.

Returns the *plane-mask* of the specified graphics context.

The plane-mask attribute of a graphics context specifies which bit planes of the destination drawable are modified during a graphic operation. The plane-mask is a pixel value in which a 1 bit means that the corresponding bit plane will be modified and a 0 bit means that the corresponding bit plane will not be affected during a graphic operations. Thus, the actual result of a graphic operation depends on both the function and plane-mask attributes of the graphics context and is given by the following expression:

```
(logior (logand
          (boole function source destination)
         plane-mask)

        (logandc2
         destination
          plane-mask))
```

gcontext — A **gcontext**.

gcontext-plist *gcontext* Function

Returns:
gcontext-p — Type **list**.

Returns and (with **setf**) sets the property list for the specified *gcontext*. This function provides a hook where extensions can add data.

gcontext — A **gcontext**.

gcontext-stipple *gcontext* Function

Returns:
stipple — Type **pixmap**.

Returns the *stipple* of the specified graphics context.

The stipple attribute of a graphics context is a bitmap used to prevent certain pixels in the destination of graphics operations from being affected by tiling.

The stipple and tile have the same origin. This origin point is interpreted relative to the origin of whatever destination drawable is specified in a graphics request. The stipple pixmap must have depth one and must have the same root as the graphics context. The tile pixmap must have the same root and depth as the graphics context. For stipple operations where the fill-style is **:stippled** (but not **:opaque-stippled**), the stipple pattern is tiled in a single plane and acts as an additional clip mask to be **and**ed with the clip-mask. Any size pixmap can be used for stipple or tile, although some sizes may be faster to use than others.

Specifying a pixmap for stipple or tile in a graphics context might or might not result in a copy being made. If the pixmap is later used as the destination for a graphics operation, the change might or might not be reflected in the graphics context. If the pixmap is used both as the destination for a graphics operation and as a stipple or tile, the results are not defined.

Some displays have hardware support for tiling or stippling with patterns of specific sizes. Tiling and stippling operations that restrict themselves to those sizes may run much faster than such operations with arbitrary size patterns. CLX provides functions to determine the best size for stipple or tile (see **query-best-stipple** and **query-best-tile**).

gcontext — A **gcontext**.

gcontext-subwindow-mode *gcontext* Function

Returns:
subwindow-mode — One of **:clip-by-children** or **:include-inferiors**.

Returns and (with **setf**) changes the subwindow-mode attribute of the specified graphics context.

The subwindow-mode attribute of a graphics context specifies whether subwindows obscure the contents of their parent window during a graphics operation. For a value of **:clip-by-children**, both source and destination windows are clipped by all viewable **:input-output** class children. This clipping is in addition to the clipping provided by the clip-mode attribute. For a value of **:include-inferiors**, neither the source nor destination window is clipped by its inferiors. This results in the inclusion of subwindow contents in the source and the drawing through of subwindow boundaries of the destination. The use of **:include-inferiors** on a window of one depth with mapped inferiors of differing depth is not illegal, but the semantics are not defined by the core protocol.

gcontext — A **gcontext**.

gcontext-tile *gcontext* Function

Returns:
tile — Type **pixmap**.

Returns the *tile* of the specified graphics context.

The tile attribute is a pixmap used to fill in areas for graphics operations. It is so named because copies of it are laid out side by side to fill the area.

The stipple and tile have the same origin. This origin point is interpreted relative to the origin of whatever destination drawable is specified in a graphics request. The stipple pixmap must have depth one and must have the same root as the graphics context. The tile pixmap must have the same root and depth as the graphics context. For stipple operations where the fill-style is **:stippled** (but not **:opaque-stippled**), the stipple pattern is tiled in a single plane and acts as an additional clip mask to be **and**ed with the clip-mask. Any size pixmap can be used for stipple or tile, although some sizes may be faster to use than others.

Specifying a pixmap for stipple or tile in a graphics context might or might not result in a copy being made. If the pixmap is later used as the destination for a graphics operation, the change might or might not be reflected in the graphics context. If the pixmap is used both as the destination for a graphics operation and as a stipple or tile, the results are not defined.

Some displays have hardware support for tiling or stippling with patterns of specific sizes. Tiling and stippling operations that restrict themselves to those sizes may run much faster than such operations with arbitrary size patterns. CLX provides functions to determine the best size for stipple or tile (see **query-best-stipple** and **query-best-tile**).

gcontext — A **gcontext**.

gcontext-ts-x *gcontext* Function

Returns:
ts-x — Type **int16**.

Returns the *ts-x* attribute of the specified graphics context.

The ts-x and ts-y attributes of a graphics context are the coordinates of the origin for tile pixmaps and the stipple.

gcontext — A **gcontext**.

gcontext-ts-y *gcontext* Function

Returns:
ts-y — Type **int16**.

Returns the *ts-y* attribute of the specified graphics context.

The ts-x and ts-y attributes of a graphics context are the coordinates of the origin for tile pixmaps and the stipple.

gcontext — A **gcontext**.

query-best-stipple *width height drawable* Function

Returns:
best-width, *best-height* — Type **card16**.

Returns the *best-width* and *best-height* for stipple pixmaps on the *drawable*.

The *drawable* indicates the screen and possibly the window class and depth. An **:input-only** window cannot be specified as the *drawable*. The size is returned as width and height values.

width, *height* — Specifies the width and height of the desired stipple pattern.

drawable — A **drawable**.

query-best-tile *width height drawable* Function

Returns:
best-width, *best-height* — Type **card16**.

Returns the *best-width* and *best-height* for tile pixmaps on the *drawable*.

The *drawable* indicates the screen and possibly the window class and depth. An **:input-only** window cannot be specified as the *drawable*. The size is returned as width and height values.

width, *height* — Specifies the width and height of the desired tile pattern.

drawable — A **drawable**.

Copying Graphics Contexts

5.3 CLX provides functions to copy some or all attribute values from one graphics context to another. These functions are generally more efficient than using **setf** to copy **gcontext** attributes individually.

copy-gcontext *source destination* Function

Copies all the values of the attributes of the source graphics context into the destination graphics context. The source and destination graphics contexts must have the same root and depth.

source — The source **gcontext**.

destination — The destination **gcontext**.

copy-gcontext-components *source destination* &rest *keys* Function

Copies the values of the specified attributes of the source graphics context to the destination graphics context. The source and destination graphics contexts must have the same root and depth.

source — The source **gcontext**.

destination — The destination **gcontext**.

keys — The remaining arguments are keywords, of type **gcontext-key**, which specify which attributes of the graphics context are to be copied.

Destroying Graphics Contexts

5.5 To destroy a graphics context, use **free-gcontext.**

free-gcontext *gcontext* Function

Deletes the association between the assigned resource ID and the specified graphics context, and then destroys the graphics context.

gcontext — A **gcontext**.

Graphics Context Cache

5.6 CLX provides a set of functions to control the automatic graphics context caching mechanism.

force-gcontext-changes *gcontext* Function

Forces any delayed changes to the specified graphics context to be sent out to the server. Note that **force-gcontext-changes** is called by all of the graphics functions.

gcontext — A **gcontext**.

with-gcontext *gcontext* &key **:arc-mode :background :cap-style :clip-mask :clip-ordering :clip-x :clip-y :dashes :dash-offset :exposures :fill-rule :fill-style :font :foreground :function :join-style :line-style :line-width :plane-mask :stipple :subwindow-mode :tile :ts-x :ts-y** &allow-other-keys &body *body* Macro

Changes the indicated graphics context components to the specified values only within the dynamic extent of the body. **with-gcontext** works on a per-process basis in a multiprocessing environment. The *body* is not surrounded by a **with-display** form. If there is no local cache for the specified graphics context, or if some of the component states are unknown, **with-gcontext** does the save and restore by creating a temporary graphics context and copying components to and from it using **copy-gcontext-components**.

gcontext — A **gcontext**.

:arc-mode, :background, **:cap-style**, **:clip-mask**, **:clip-ordering**, **:clip-x**, **:clip-y**, **:dashes**, **:dash-offset**, **:exposures**, **:fill-rule**, **:fill-style**, **:font**, **:foreground**, **:function**, **:join-style**, **:line-style**, **:line-width**, **:plane-mask**, **:stipple**, **:subwindow-mode**, **:tile**, **:ts-x**, **:ts-y** —These keyword arguments and associated values specify which graphics context components are to be changed. Any components not specified are left unmodified. See paragraph 5.2, Creating Graphics Contexts, for more information.

body — The body of code which will have access to the altered graphics context.

GRAPHIC OPERATIONS

6

Introduction

6.1 Once connected to an X server, a client can use CLX functions to perform graphic operations on drawables.

This section describes CLX functions to:

- Operate on areas and planes
- Draw points
- Draw lines
- Draw rectangles
- Draw arcs
- Draw text

Area and Plane Operations

6.2 clear-area clears an area or an entire window to the background. Since pixmaps do not have backgrounds, they cannot be filled by using the functions described in the following paragraphs. Instead, you should use **draw-rectangle**, which sets the pixmap to a known value. See paragraph 6.5, Drawing Rectangles, for information on **draw-rectangle**.

clear-area *window* &key (**:x** 0) (**:y** 0) **:width :height :exposures-p** Function

Draws a rectangular area in the specified *window* with the background pixel or pixmap of the *window*. The **:x** and **:y** coordinates are relative to the *window* origin, and specify the upper-left corner of the rectangular area that is to be cleared. A **nil** or zero value for **:height** or **:width** clears the remaining area (height – y or width – x). If the *window* has a defined background tile, the rectangle is tiled by using a plane-mask of all ones and a function of **:copy**. If the *window* has background **:none**, the contents of the *window* are not changed. In either case, if **:exposures-p** is non-**nil**, then one or more **:exposure** events are generated for regions of the rectangle that are either visible or are being retained in a backing store.

To clear the entire area in a specified *window*, use (**clear-area** *window*).

window — A **window**.

:x, **:y** — Upper-left corner of the area to be cleared. These coordinates are relative to the *window* origin. Type is **int16**.

:width — The width of the area to clear or **nil** to clear to the remaining width of the window. Type is **card16** or **null**.

:height — The height of the area to clear or **nil** to clear to the remaining height of the window. Type is **card16** or **null**.

:exposures-p — Specifies if **:exposure** events should be generated for the affected areas. Type **boolean**.

copy-area *source gcontext source-x source-y width height destination destination-x destination-y* Function

Copies the specified rectangular area from the *source* **drawable** to the specified rectangular area of the *destination* **drawable**, combining them as specified in the supplied graphics context (*gcontext*). The *x* and *y* coordinates are relative to their respective drawable origin, with each pair specifying the upper left corner of the area.

If either regions of the *source* area are obscured and have not been retained in backing store, or regions outside the boundaries of the *source* **drawable** are specified, those regions are not copied. Instead, the following occurs on all corresponding *destination* regions that are either visible or are retained in backing store:

- If the *destination* rectangle is a window with a background other than **:none**, these corresponding regions of the *destination* are tiled, using plane-mask of all ones and function of **boole-1** (copy source), with that background.
- If the exposures attribute of the graphics context is **:on**, then **:graphics-exposure** events for all corresponding *destination* regions are generated (regardless of tiling or whether the *destination* is a window or a pixmap).
- If exposures is **:on** but no regions are exposed, a **:no-exposure** event is generated. Note that by default, exposures is **:on** for new graphics contexts. See Section 5, Graphics Contexts, for further information.

source — Source **drawable**.

gcontext — The graphics context to use during the copy operation.

source-x, *source-y* — The x and y coordinates of the upper-left corner of the area in the *source* **drawable**. These coordinates are relative to the *source* **drawable** origin. Type is **int16**.

width, *height* — The width and height of the area being copied. These apply to both the *source* and *destination* areas. Type is **card16**.

destination — The destination **drawable**.

destination-x, *destination-y* — The x and y coordinates of the upper left corner of the area in the *destination* **drawable**. These coordinates are relative to the *destination* **drawable** origin. Type is **int16**.

copy-plane *source gcontext plane source-x source-y width height destination destination-x destination-y* Function

Uses a single bit plane of the specified rectangular area of the *source* **drawable** along with the specified graphics context (*gcontext*) to modify the specified rectangle area of the *destination* **drawabl**e. The drawables specified by the *source* and *destination* arguments must have the same root but need not have the same depth.

Effectively, this operation forms a pixmap of the same depth as *destination* and with a size specified by the *source* area. It then uses the foreground and background from the graphics context (foreground where the bit-plane in *source* contains a one bit, background where the bit-plane in *source* contains a zero bit), and the equivalent of a **copy-area** operation is performed with all the same exposure semantics. This can also be thought of as using the specified region of the *source* bit-plane as a stipple with a fill-style of **:opaque-stippled** for filling a rectangular area of the *destination*.

source — The source **drawable**.

gcontext — The graphics context to use during the copy operation.

plane — Specifies the bit-plane of the *source* **drawable**. Exactly one bit must be set. Type is **pixel**.

source-x, *source-y* — The *x* and *y* coordinates of the upper-left corner of the area in the *source* **drawable**. These coordinates are relative to the *source* **drawable** origin. Type is **int16**.

width, *height* — The width and height of the area being copied. These apply to both the *source* and *destination* areas. Type is **card16**.

destination — The destination **drawable**.

destination-x, *destination-y* — The x and y coordinates of the upper-left corner of the destination area in the *destination* **drawable**. These coordinates are relative to the *destination* **drawable** origin. Type is **int16**.

Drawing Points

6.3 The **draw-point** and **draw-points** functions make use of the following graphics context components: function, plane-mask, foreground, subwindow-mode, clip-x, clip-y, clip-ordering, clip-region and clip-mask.

The **draw-point** function uses the foreground pixel and function components of the graphics context to draw a single point into the specified drawable, while **draw-points** draws multiple points into the specified drawable. These functions are not affected by the tile or stipple in the graphics context.

draw-point *drawable gcontext x y* — Function

Combines the foreground pixel in the *gcontext* with the pixel in the *drawable* specified by the *x* and *y* coordinates.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing the point.

x, *y* — The *x* and *y* coordinates of the point drawn. Type is **int16**.

draw-points *drawable gcontext points* &optional *relative-p* — Function

Combines the foreground pixels in the graphics context with the pixels at each point in the *drawable*. The points are drawn in the order listed.

draw-points requires a mode argument, *relative-p* that indicates whether the points are relative to the destination origin or to the previous point. In either case, the first point is always relative to the destination origin. The rest of the points are relative either to the *drawable*'s origin or to the previous point, depending on the value of *relative-p*.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing the points.

points — A list of points to be drawn in the order listed. The first point is always relative to the *drawable*'s origin; if *relative-p*, the rest of the points are drawn relative to the previous point, else they are drawn relative to the *drawable*'s origin. Type is **point-seq**.

relative-p — Specifies the coordinate mode used for drawing the pixels either relative to the origin or to the previous point. Type **boolean**.

Drawing Lines

6.4 The **draw-line**, **draw-lines**, and **draw-segments** functions use the following graphics context components: background, cap-style, clip-x-origin, clip-y-origin, clip-mask, dash-list, dash-offset, fill-style, foreground, function, plane-mask, line-width, line-style, stipple, subwindow-mode, tile, ts-x-origin, and ts-y-origin.

The **draw-lines** function also uses the join-style graphics context component.

draw-line *drawable gcontext x1 y1 x2 y2* &optional *relative-p* Function

Draws a line from the point *x1*,*y1* to the point *x2*,*y2*. When *relative-p* is true, the first point is relative to the destination origin but the second point is relative to the first point. When *relative-p* is **nil**, both points are relative to the destination origin.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing the line.

x1, *y1*, *x2*, *y2* — The end points of the line.

relative-p — Specifies the coordinate mode used for drawing the line either relative to the origin or the previous point. In either case, the first point is always drawn relative to the *drawable*'s origin.

draw-lines *drawable gcontext points* &key **:relative-p :fill-p** (**:shape :complex**) Function

Draws a line between each pair of *points* in the points list. The lines are drawn in the order listed and join correctly at all intermediate points. The join-style graphics context component defines the type of joint to use. When the first and last points coincide, the first and last lines also join correctly to produce a hollow polygon.

When **:relative-p** is true, the first point is always relative to the destination origin, but the rest are relative to the previous point. When **:relative-p** is **nil**, the rest of the points are drawn relative to the destination origin.

When **:fill-p** is true, the polygon defined by the *points* list is filled. The **:shape** keyword provides the server with a hint about how to fill the polygon. **:shape** can be either **:complex** (by default), **:convex**, or **:non-convex**.

The **:convex** operand is the simplest type of area and the fastest to fill. A fill area is convex if every straight line connecting any two interior points is entirely inside the area. For example, triangles and rectangles are convex polygons.

The **:non-convex** operand is for filling an area that is not convex and is also not self-intersecting. Filling this type of area is harder than filling a convex area, but easier than filling one that is self-intersecting. For example, the shape of the letter "T" is non-convex and non-self-intersecting.

The **:complex** operand is the most general (and therefore the hardest) type of fill area. A complex fill area can be non-convex and self-intersecting. For example, draw the outline of a bow tie, without lifting your pencil or tracing over an edge twice. This shape is non-convex and intersects itself at the knot in the middle.

NOTE: Unless you are sure that a shape is **:convex** or **:non-convex**, it should always be drawn as a **:complex** shape. If **:convex** or **:non-convex** is specified incorrectly, the graphics result is undefined.

---

drawable — The destination **drawable**.

gcontext — The graphics context for drawing the lines.

points — A list of points that define the lines. Type is **point-seq**.

:relative-p — The coordinate mode of the points.

:fill-p — When true, a filled polygon is drawn instead of a polyline.

:shape — A hint that allows the server to use the most efficient area fill algorithm. Either **:convex**, **:non-convex**, or **:complex**.

draw-segments *drawable gcontext segments* Function

Draws multiple lines, not necessarily connected. *segments* is a sequence of the form {x1 y1 x2 y2}*, in which each subsequence specifies the end points of a line segment. Line segments are drawn in the order given by *segments*. Unlike **draw-lines**, no joining is performed at coincident end points.

drawable — The destination **drawable** to receive the line segments.

gcontext — Specifies the graphics context for drawing the lines.

segments — The points list for the segments to draw. Type is **seq**.

Drawing Rectangles

6.5 The **draw-rectangle** and **draw-rectangles** functions draw hollow or filled outlines of the specified rectangle or rectangles as if a five-point polyline were specified for each rectangle, as follows:

[x,y,] [x+width,y] [x+width,y+height] [x,y+height] [x,y]

draw-rectangle and **draw-rectangles** use the following graphics context components: background, function, plane-mask, foreground, subwindow-mode, cap-style, clip-x, clip-y, clip-ordering, clip-region and clip-mask, dash-list, dash-offset, fill-style, join-style, line-width, line-style, stipple, tile, ts-x-origin, and ts-y-origin.

draw-rectangle *drawable gcontext x y width height* &optional *fill-p* Function

Draws a rectangle defined by the *x*, *y*, *width*, and *height* arguments.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing the rectangle.

x, *y* — The x and y coordinates that define the upper left corner of the rectangle. The coordinates are relative to the destination origin. Type is **int16**.

width, *height* — Specifies the width and height that define the outline of the rectangle. Type is **card16**.

fill-p — Specifies whether the rectangle is filled or not. Type **boolean**.

draw-rectangles *drawable gcontext rectangles* &optional *fill-p* Function

Draws the rectangles in the order listed in *rectangles*. For the specified *rectangle* or *rectangles*, no pixel is drawn more than once. The x and y coordinates of each rectangle are relative to the destination origin and define the upper left corner of the rectangle. If rectangles intersect, the intersecting pixels are drawn multiple times.

drawable — The destination **drawable**.

gcontext — The graphics context.

rectangles — A list specifying the upper left corner x and y, width and height of the rectangles. Type is **rect-seq**.

fill-p — Specified if the rectangles are filled or not. Type is **boolean**.

Drawing Arcs

6.6 **draw-arc** draws a single circular or an elliptical arc, while **draw-arcs** draws multiple circular or elliptical arcs. **draw-arc** and **draw-arcs** use the following graphics context components: arc-mode, background, cap-style, clip-x, clip-y, clip-mask, dash-list, dash-offset, fill-style, foreground, join-style, function, plane-mask, line-width, line-style, stipple, subwindow-mode, tile, ts-x-origin, and ts-y-origin.

draw-arc *drawable gcontext x y width height angle1 angle2* &optional *fill-p* Function

Draws either a circular or an elliptical arc. Also, outlined or filled arcs can be drawn. Each arc is specified by a rectangle (*x*, *y*, *width*, and *height*) and two angles (*angle1* and *angle2*). The angles are signed integers in radians, with positive indicating counterclockwise motion and negative indicating clockwise motion. The start of the arc is specified by *angle1*, and the path and extent of the arc is specified by *angle2* relative to the start of the arc. If the magnitude of *angle2* is greater than 360 degrees, it is truncated to 360 degrees. The *x* and *y* coordinates of the rectangle are relative to the *drawable*'s origin.

For example, an arc specified as [*x*,*y*,*width*,*height*,*angle1*,*angle2*] has the origin of the major and minor axes at:

[*x*+(*width*/2),*y*+(*height*/2)]

The infinitely thin path describing the entire circle/ellipse intersects the horizontal axis at:

[*x*,*y*+(*height*/2)] and [*x*+*width*,*y*+(*height*/2)]

The intersection of the vertical axis is at:

[*x*+(*width*/2),*y*] and [*x*+(*width*/2),*y*+*height*]

These coordinates can be fractional; that is, they are not truncated to discrete coordinates. Note that the angle values are slightly different in CLX than in the X protocol specification.

If *fill-p* is **nil**, then only the outline of the arc is drawn. Otherwise, if *fill-p* is true, **draw-arc** fills the area bounded by the arc outline and one or two line segments, depending on the arc-mode. If the arc-mode is **:chord**, the filled area is bounded by the arc outline and the line segment joining the arc end points. If the arc-mode is **:pie-slice**, the filled area is bounded by the arc outline and the two line segments joining each arc end point with the center point.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing the arc.

x, *y* — The x and y coordinates of the arc rectangle relative to the origin of the *drawable*. Type is **int16**.

width, *height* — Specifies the width and height of the rectangle. These are the major and minor axes of the arc. Type is **card16**.

angle1 — Specifies the start of the arc in radians. Type is **angle**.

angle2 — Specifies the direction and end point of the arc. Type is **angle**.

fill-p — Specifies whether the arc is filled or not. Type **boolean**.

draw-arcs *drawable gcontext arcs* &optional *fill-p* Function

Draws circular or elliptical, outlined or filled arcs. Each arc is specified by a rectangle and two angles. For a more detailed description, see **draw-arc**.

The arcs are filled in the order listed. For any given arc, no pixel is drawn more than once. If regions intersect, the intersecting pixels are drawn multiple times.

drawable — Specifies the **drawable** where you want the arcs drawn.

gcontext — Specifies the graphics context for drawing the arc.

arcs — A sequence containing the width, height, angle1, and angle2 arguments defining the arcs. See **draw-arc** for more detail. Type is **arc-seq**.

fill-p — Specifies whether the arcs are filled or not. Type is **boolean**.

Drawing Text

6.7 CLX provides functions for drawing text using text fonts provided by the X server. An X font is array of character bit maps indexed by integer codes. See Section 8 for a complete discussion of the CLX functions used to manage fonts and characters.

Since Common Lisp programs typically represent text as sequences of characters (that is, strings), CLX text functions must be prepared to convert a Common Lisp character into the integer code used to index the appropriate character bitmap in a given font. The **:translate** argument to a text function is a function which performs this conversion. The default **:translate** function handles all characters that satisfy **graphic-char-p** by converting each character into its ASCII code. Note that the assumption made by the default **:translate** function—that is, that an X font indexes bitmaps by ASCII codes—is often valid, but other encodings are possible. In general, a **:translate** function can perform complex transformations. It can be used to convert non-character input, to handle non-ASCII character encodings, and to change the fonts used to access character bitmaps. The complete behavior of a **:translate** function is given below by describing a prototypical **translate-function**.

CLX offers two different ways to draw text—filled text and block text. The **draw-glyph** and **draw-glyphs** functions create filled text, in which each character image is treated as an area to be filled according to the fill-style of the given graphics context, without otherwise disturbing the surrounding background. In addition, filled text sends a complex type of server request which allows a series of font indices, font changes, and horizontal position changes to be compiled into a single request. Filled text functions use the following graphics context attributes: background, clip-mask, clip-x-origin, clip-y-origin, fill-style, font, foreground, function, plane-mask, stipple, subwindow-mode, tile, ts-x-origin, ts-y-origin.

Block text is a rendering style commonly used by display terminals, in which each character image appears in the foreground pixel inside a rectangular character cell drawn in the graphics context background pixel. The **draw-image-glyph** and **draw-image-glyphs** functions create block text. Block text functions use the following graphics context attributes: background, clip-mask, clip-x-origin, clip-y-origin, font, foreground, plane-mask, stipple, subwindow-mode, tile, ts-x-origin, ts-y-origin.

draw-glyph *drawable gcontext x y element* &key **:translate :width** (**:size :default**) Function

Returns:
output-p — Type **boolean**.
width — Type **int32** or **null**.

Draws a single character of filled text represented by the given *element*. The given *x* and *y* specify the left baseline position for the character. The first return value is true if the character is successfully translated and drawn, or **nil** if the **:translate** function did not translate it. The second return value gives the total pixel width of the character actually drawn, if known.

Specifying a **:width** is a hint to improve performance. The **:width** is assumed to be the total pixel width of the character actually drawn. Specifying **:width** permits appending the output of subsequent calls to the same protocol request, provided *gcontext* has not been modified in the interim. If **:width** is not specified, appending of subsequent output might not occur (unless **:translate** returns the character width).

The **:size** specifies the element size of the destination buffer given to **:translate** (either 8, 16, or **:default**). If **:default** is specified, the size is based on the current font, if known; otherwise, 16 is used.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing text.

x, *y* — The left baseline position for the character drawn.

element — A character or other object to be translated into a font index.

:translate — A function to translate text to font indexes. Default is **#'translate-default**.

:width — The total pixel width of the character actually drawn, if known.

:size — Specifies the element size of the destination buffer given to **:translate** (8, 16, or **:default**).

draw-glyphs *drawable gcontext x y sequence* &key (**:start** 0) **:end :translate :width** (**:size :default**) Function

Returns:
new-start — Type **array-index** or **null**.
width — Type **int32** or **null**.

Draws the filled text characters represented by the given sequence. **:start** and **:end** define the elements of the sequence which are drawn. The given *x* and *y* specify the left baseline position for the first character. The first return value is **nil** if all characters are successfully translated and drawn; otherwise, the index of the first untranslated sequence element is returned. The second return value gives the total pixel width of the characters actually drawn, if known.

Specifying a **:width** is a hint to improve performance. The **:width** is assumed to be the total pixel width of the character sequence actually drawn. Specifying **:width** permits appending the output of subsequent calls to the same protocol request, provided *gcontext* has not been modified in the interim. If **:width** is not specified, appending of subsequent output might not occur (unless **:translate** returns the character width).

The **:size** specifies the element size of the destination buffer given to **:translate** (either 8, 16, or **:default**). If :**default** is specified, the size is based on the current font, if known; otherwise, 16 is used.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing text.

x, *y* — The left baseline position for the character drawn.

sequence — A sequence of characters or other objects to be translated into font indexes.

:start, **:end** — Start and end indexes defining the elements to draw.

:translate — A function to translate text to font indexes. Default is **#'translate-default**.

:width — The total total pixel width of the character actually drawn, if known.

:size — The element size of the destination buffer given to **:translate** (8, 16, or **:default**).

draw-image-glyph *drawable gcontext x y element* &key **:translate :width** (**:size :default**) Function

Returns:
output-p — Type **boolean**.
width — Type **int32** or **null**.

Draws a single character of block text represented by the given *element*. The given *x* and *y* specify the left baseline position for the character. The first return value is true if the character is successfully translated and drawn, or **nil** if the **:translate** function did not translate it. The **:translate** function is allowed to return an initial font change. The second return value gives the total pixel width of the character actually drawn, if known.

The **:translate** function may not return a horizontal position change, since **draw-image-glyph** does not generate complex output requests.

Specifying a **:width** is a hint to improve performance. The **:width** is assumed to be the total pixel width of the character actually drawn. Specifying **:width** permits appending the output of subsequent calls to the same protocol request, provided *gcontext* has not been modified in the interim. If **:width** is not specified, appending of subsequent output might not occur (unless **:translate** returns the character width).

The **:size** specifies the element size of the destination buffer given to **:translate** (either 8, 16, or **:default**). If **:default** is specified, the size is based on the current font, if known; otherwise, 16 is used.

drawable — The destination **drawable**.

gcontext — The graphics context for drawing text.

x, *y* — The left baseline position for the character drawn.

element — A character or other object to be translated into a font index.

:translate — A function to translate text to font indexes. Default is **#'translate-default**.

:width — The total pixel width of the character actually drawn, if known.

:size — Specifies the element size of the destination buffer given to **:translate** (8, 16, or **:default**).

draw-image-glyphs *drawable gcontext x y sequence* &key (**:start** 0) **:end** **:translate** **:width** (**:size** **:default**) — Function

Returns:
new-start — Type **array-index** or **null**.
width — Type **int32** or **null**.

Draws the block text characters represented by the given **sequence**. **:start** and **:end** define the elements of the *sequence* which are drawn. The given *x* and *y* specify the left baseline position for the first character. The first return value is **nil** if all characters are successfully translated and drawn; otherwise, the index of the first untranslated sequence element is returned. The **:translate** function is allowed to return an initial font change. The second return value gives the total pixel width of the characters actually drawn, if known.

The **:translate** function may not return a horizontal position change, since **draw-image-glyphs** does not generate complex output requests.

Specifying a **:width** is a hint to improve performance. The **:width** is assumed to be the total pixel width of the character sequence actually drawn. Specifying **:width** permits appending the output of subsequent calls to the same protocol request, provided *gcontext* has not been modified in the interim. If **:width** is not specified, appending of subsequent output might not occur (unless **:translate** returns the character width).

The **:size** specifies the element size of the destination buffer given to **:translate** (either 8, 16, or **:default**). If **:default** is specified, the size will be based on the current font, if known; otherwise, 16 is used.

drawable — The destination **drawable**.

x, *y* — The left baseline position for the character drawn.

gcontext — The graphics context for drawing text.

sequence — A sequence of characters or other objects to be translated into font indexes.

:start, **:end** — Start and end indexes defining the elements to draw.

:translate — A function to translate text to font indexes. Default is **#'translate-default**.

:width — The total total pixel width of the character actually drawn, if known.

:size — The element size of the destination buffer given to **:translate** (8, 16, or **:default**).

translate-function *source source-start source-end font destination destination-start* — Function

Returns:
first-not-done — Type **array-index**.
to-continue — Type **int16**, **font**, or **null**.
current-width — Type **int32** or **null**.

A function used as the **:translate** argument for text functions. Converts elements of the *source* (sub)sequence into font indexes for the given *font* and stores them into the *destination* vector.

The *destination* vector is created automatically by CLX. *destination* is guaranteed to have room for (– *source-end source-start*) integer elements, starting at *destination-start*. Elements of *destination* can be either **card8** or **card16** integers, depending on the context. *font* is the current font, if known, or **nil** otherwise. Starting with the element at *source-start*, **translate-function** should translate as many elements of *source* as possible (up to the *source-end* element) into indexes in the current *font*, and store them into *destination*. The first return value should be the source index of the first untranslated element.

The second return value indicates the changes which should be made to the current text output request before translating the remaining *source* elements. If no further elements need to be translated, the second return value should be **nil**. If a horizontal motion is required before further translation, the second return value should be the change in x position. If a font change is required for further translation, the second return value should be the new font.

If known, the pixel width of the translated text can be returned as the third value; this can allow for appending of subsequent output to the same protocol request, if no overall width has been specified at the higher level.

source — A sequence of characters or other objects to be translated.

source-start — An array-index specifying the first *source* element to be translated.

source-end — An array-index specifying the end of the *source* subsequence to be translated.

font — The font indexed by translated *source* elements.

destination — A vector where translated *source* elements are stored.

destination-start — An array-index specifying the position to begin storing translated *source* elements.

IMAGES

7

Introduction

7.1 The X protocol provides for the transfer of images (two-dimensional arrays of pixel data) between a client program and a **drawable**. The format for image data can vary considerably. In order to present a uniform data representation for the manipulation of a variety of images, CLX defines a special **image** data type. Additional **image** subtypes — **image-xy** and **image-z** — allow for the representation of an image either as a sequence of bit planes or as an array of pixels. CLX includes functions for accessing **image** objects; for transferring image data between **image** objects, **drawables**, and files; and also for direct transfer of raw image data.

Image Types

7.2 The **image** data type is the base type for all **image** objects. **image-xy** and **image-z** are subtypes of the **image** type which furnish accessors specialized for different image representations.

Basic Images

7.2.1 The following paragraphs describe the CLX functions that can be used to access all types of **image** objects.

image-blue-mask *image* — Function

Returns:
mask — Type **pixel** or **null**.

Returns (and with **setf**) changes the *mask* that selects the pixel subfield for blue intensity values. The *mask* is non-**nil** only for images for **:direct-color** or **:true-color** visual types.

image — An **image** object.

image-depth *image* — Function

Returns:
depth — Type **card8**.

Returns the *depth* (that is, the number of bits per pixel) for the *image*.

image — An **image** object.

image-green-mask *image* — Function

Returns:
mask — Type **pixel** or **null**.

Returns (and with **setf**) changes the mask that selects the pixel subfield for green intensity values. The mask is non-**nil** only for images for **:direct-color** or **:true-color** visual types.

image — An **image** object.

image-height *image* Function

Returns:
height — Type **card16**.

Returns the *height* of the *image* in pixels.

image — An **image** object.

image-name *image* Function

Returns:
name — Type **stringable** or **null**.

Returns and (with **setf**) changes the *name* string optionally associated with the *image*.

image — An **image** object.

image-plist *image* Function

Returns:
plist — Type **list**.

Returns and (with **setf**) changes the *image* property list. The property list is a hook for added application extensions.

image — An **image** object.

image-red-mask *image* Function

Returns:
mask — Type **pixel** or **null**.

Returns (and with **setf**) changes the *mask* which selects the pixel subfield for red intensity values. The *mask* is non-**nil** only for images for **:direct-color** or **:true-color** visual types.

image — An **image** object.

image-width *image* Function

Returns:
width — Type **card16**.

Returns the *width* of the *image* in pixels.

image — An **image** object.

image-x-hot *image* Function

Returns:
x-position — Type **card16** or **null**.

Returns and (with **setf**) changes the x position of the hot spot for an image used as a cursor glyph. The hot spot position is specified relative to the upper-left origin of the *image*.

image — An **image** object.

image-y-hot *image* Function

Returns:
y-position — Type **card16** or **null**.

Returns and (with **setf**) changes the y position of the hot spot for an image used as a cursor glyph. The hot spot position is specified relative to the upper-left origin of the *image*.

image — An **image** object.

XY-Format Images

7.2.2 The **image-xy** subtype represents an image as a sequence of bitmaps, one for each plane of the image, in most-significant to least-significant bit order. The following paragraphs describe the additional CLX functions that can be used to access **image-xy** objects.

image-xy-bitmap-list *image* Function

Returns:
bitmaps — Type **list** of **bitmap**.

Returns and (with **setf**) changes the list of bitmap planes for the *image*.

image — An **image-xy** object.

Z-Format Images

7.2.3 The **image-z** subtype represents an image as a two-dimensional array of pixels, in scanline order. The following paragraphs describe the additional CLX functions that can be used to access **image-z** objects.

image-z-bits-per-pixel *image* Function

Returns:
pixel-data-size — One of 1, 4, 8, 16, 24, or 32.

Returns and (with **setf**) changes the number of bits per data unit used to contain a pixel value for the *image*. Depending on the storage format for image data, this value can be larger than the actual *image* depth.

image — An **image-z** object.

image-z-pixarray *image* Function

Returns:
pixarray — Type **pixarray**.

Returns and (with **setf**) changes the two-dimensional array of pixel data for the *image*.

image — An **image-z** object.

Image Functions

7.3 The following paragraphs describe the CLX functions used to:

- Create an **image** object.
- Copy an image or a subimage.
- Read an image from a **drawable**.
- Display an image to a **drawable**.

create-image &key **:bit-lsb-first-p :bits-per-pixel :blue-mask :byte-lsb-first-p :bytes-per-line :data :depth :format :green-mask :height :name :plist :red-mask :width :x-hot :y-hot** Function

Returns:
image — Type **image**.

Creates an **image** object from the given **:data** and returns either an **image**, **image-xy**, or an **image-z**, depending on the type of image **:data**. If the **:data** is a list, it is assumed to be a **list** of **bitmaps** and an **image-xy** is created. If the **:data** is a **pixarray**, an **image-z** is created. Otherwise, the **:data** must be an array of bytes (**card8**), in which case a basic **image** object is created.

If the **:data** is a list, each element must be a bitmap of equal size. **:width** and **:height** default to the bitmap width — (**array-dimension bitmap** 1) — and the bitmap height — (**array-dimension bitmap** 0) — respectively. **:depth** defaults to the number of bitmaps.

If the **:data** is a **pixarray**, **:width** and **:height** default to the **pixarray** width — (**array-dimension pixarray** 1), and the pixarray height — (**array-dimension pixarray** 0), respectively. **:depth** defaults to (**pixarray-depth :data**). The **:bits-per-pixel** is rounded to a valid size, if necessary. By default, the **:bits-per-pixel** is equal to the **:depth**.

If the **:data** is an array of **card8**, the **:width** and **:height** are required to interpret the image data correctly. The **:bits-per-pixel** defaults to the **:depth**, and the **:depth** defaults to 1. **:bytes-per-line** defaults to:

(**floor** (**length :data**) (* **:bits-per-pixel :height**))

The **:format** defines the storage format of image data bytes and can be one of the following values:

- **:xy-pixmap** — The **:data** is organized as a set of bitmaps representing image bit planes, appearing in most-significant to least-significant bit order.
- **:z-pixmap** — The **:data** is organized as a set of pixel values in scanline order.
- **:bitmap** — Similar to **:xy-pixmap**, except that the **:depth** must be 1, and 1 and 0 bits represent the foreground and background pixels, respectively.

By default, the **:format** is **:bitmap** if **:depth** is 1; otherwise, **:z-pixmap**.

:bit-lsb-first-p — For a returned image, true if the order of bits in each **:data** byte is least-significant bit first.

:bits-per-pixel — One of 1, 4, 8, 16, 24, or 32.

:blue-mask — For **:true-color** or **:direct-color** images, a pixel mask.

:byte-lsb-first-p — For a returned *image*, true if the **:data** byte order is least-significant byte first.

:bytes-per-line — For a returned *image*, the number of **:data** bytes per scanline.

:data — Either a **list** of **bitmaps**, a **pixarray**, or an array of **card8** bytes.

:depth — The number of bits per displayed pixel.

:format — One of **:bitmap**, **:xy-format**, or **:z-format**.

:green-mask — For **:true-color** or **:direct-color** images, a pixel mask.

:height — A **card16** for the image height in pixels.

:name — An optional **stringable** for the image name.

:plist — An optional image property list.

:red-mask — For **:true-color** or **:direct-color** images, a pixel mask.

:width — A **card16** for the image width in pixels.

:x-hot — For a **cursor** image, the x position of the hot spot.

:y-hot — For a cursor image, the y position of the hot spot.

copy-image *image* &key (**:x** 0) (**:y** 0) **:width :height :result-type** Function

Returns:
new-image — Type **image**.

Returns a new image, of the given **:result-type**, containing a copy of the portion of the *image* defined by **:x**, **:y**, **:width**, and **:height**. By default, **:width** is:

(– (**image-width** *image*) **:x**)

and **:height** is:

(– (**image-height** *image*) **:y**)

If necessary, the new image is converted to the **:result-type**, that can be one of the following values:

- **'image-x** — A basic **image** object is returned.
- **'image-xy** — An **image-xy** is returned.
- **'image-z** — An **image-z** is returned.

image — An **image** object.

:x, **:y** — **card16** values defining the position of the upper-left corner of the subimage copied.

:width, **:height** — **card16** values defining the size of subimage copied.

:result-type — One of **'image-x**, **'image-xy**, or **'image-z**.

get-image *drawable* &key **:x :y :width :height :plane-mask** (**:format :z-format**) **:result-type** Function

Returns:
image — Type **image**.

Returns an *image* containing pixel values from the region of the *drawable* given by **:x**, **:y**, **:width**, and **:height**. The bits for all planes selected by 1 bits in the **:plane-mask** are returned as zero; the default **:plane-mask** is all 1 bits. The **:format** of the returned pixel values may be either **:xy-format** or **:z-format**.

The **:result-type** defines the type of image object returned:

- **'image-x** — A basic **image** object is returned.
- **'image-xy** — An **image-xy** is returned.
- **'image-z** — An **image-z** is returned.

By default, **:result-type** is **'image-z** if **:format** is **:z-format** and **'image-xy** if **:format** is **:xy-format**.

drawable — A **drawable**.

:x, **:y** — **card16** values defining the upper-left **drawable** pixel returned. These arguments are required.

:width, **:height** — **card16** values defining the size of the *image* returned. These arguments are required.

:plane-mask — A pixel mask.

:format — Either **:xy-pixmap** or **:z-pixmap**.

:result-type — One of **'image-x**, **'image-xy**, or **'image-z**.

put-image *drawable gcontext image* &key (**:src-x** 0) (**:src-y** 0) **:x :y** **:width :height :bitmap-p** — Function

Displays a region of the *image* defined by **:src-x**, **:src-y**, **:width**, and **:height** on the destination d*rawable*, with the upper-left pixel of the *image* region displayed at the *drawable* position given by **:x** and **:y**. By default, **:width** is:

(– (**image-width** *image*) **:src-x**)

and **:height** is:

(– (**image-height** *image*) **:src-y**)

The following attributes of the *gcontext* are used to display the **image**: clip-mask, clip-x, clip-y, function, plane-mask, and subwindow-mode.

The **:bitmap-p** argument applies only to images of depth 1. In this case, if **:bitmap-p** is true or if the *image* is a basic **image** object created with **:format :bitmap**, the *image* is combined with the foreground and background pixels of the **gcontext**. 1 bits of the *image* are displayed in the foreground pixel and 0 bits are displayed in the background pixel.

drawable — The destination **drawable**.

gcontext — The graphics context used to display the *image*.

image — An **image** object.

:src-x, **:src-y** — **card16** values defining the upper-left position of the *image* region to display.

:x, **:y** — The position in the *drawable* where the *image* region is displayed. These arguments are required.

:width, **:height** — **card16** values defining the size of the *image* region displayed.

:bitmap-p — If *image* is depth 1, then if true, foreground and background pixels are used to display 1 and 0 bits of the *image*.

Image Files

7.4 CLX provides functions that allow images to be written to a file in a standard X format. The following paragraphs describe the CLX functions used to:

- Read an image from a file.
- Write an image to a file.

read-bitmap-file *pathname* Function

Returns:
image — Type **image**.

Reads an image file in standard X format and returns an **image** object. The returned *image* can have depth greater than one.

pathname — An image file pathname.

write-bitmap-file *pathname image* &optional *name* Function

Writes the *image* to an image file in standard X format. The *image* can have depth greater than one. The *name* is an image identifier written to the file; the default *name* is (**or** (**image-name** *image*) **'image**).

pathname — An image file pathname.

image — An **image** object.

name — A **stringable** image name.

Direct Image Transfer

7.5 For cases where the **image** representation is not needed, CLX provides functions to read and display image data directly.

get-raw-image *drawable* &key **:data** (**:start** 0) **:x :y :width :height** **:plane-mask** (**:format :z-format**) (**:result-type '**(**vector card8**)) Function

Returns:
data — Type **sequence** or **card8**.
depth — Type **card8**.
visual — Type **card29**.

Returns a sequence of image data from the region of the *drawable* given by **:x**, **:y**, **:width**, and **:height**. If **:data** is given, it is modified beginning with the element at the **:start** index and returned. The *depth* and *visua*l type ID of the *drawable* are also returned.

The bits for all planes selected by 1 bits in the **:plane-mask** are returned as zero; the default **:plane-mask** is all 1 bits. The **:format** of the returned pixel values may be either **:xy-format** or **:z-format**. The **:result-type** defines the type of image data returned.

The calling program is responsible for handling the byte-order and bit-order returned by the server for the *drawable*'s display (see **display-byte-order** and **display-image-lsb-first-p**).

drawable — A **drawable**.

:data — An optional **sequence** of **card8**.

:start — The index of the first **:data** element modified.

:x, **:y** — **card16** values defining the size of the **image** returned. These arguments are required.

:width, **:height — card16** values defining the size of the image returned.These arguments are required.

:plane-mask — A pixel mask.

:format — Either **:xy–pixmap** or **:z–pixmap**. This argument is required.

:result-type — The type of image data sequence to return.

put-raw-image *drawable gcontext data* &key (**:start** 0) **:depth :x :y** **:width :height** (**:left-pad** 0) **:format** Function

Displays a region of the image data defined by **:start**, **:left-pad**, **:width,** and **:height** on the destination *drawable*, with the upper-left pixel of the image region displayed at the *drawable* position given by **:x** and **:y**.

The **:format** can be either **:xy-pixmap**, **:z-pixmap**, or **:bitmap**. If **:xy-pixmap** or **:z-pixmap** formats are used, **:depth** must match the depth of the destination *drawable*. For **:xy-pixmap**, the data must be in XY format. For **:z-pixmap**, the data must be in Z format for the given **:depth**.

If the **:format** is **:bitmap**, the **:depth** must be 1. In this case, the image is combined with the foreground and background pixels of the *gcontext*. 1 bits of the image are displayed in the foreground pixel and 0 bits are displayed in the background pixel.

The **:left-pad** must be zero for **:z-pixmap** format. For **:bitmap** and **:xy-pixmap** formats, the **:left-pad** must be less than the bitmap-scanline-pad for the *drawable*'s display (see **display-bitmap-format**). The first **:left-pad** bits in every scanline are to be ignored by the server; the actual image begins that many bits into the data.

The following attributes of the *gcontext* are used to display the **image**: clip-mask, clip-x, clip-y, function, plane-mask, and subwindow-mode.

The calling program is responsible for handling the byte-order and bit-order required by the server for the *drawable*'s display (see **display-byte-order** and **display-image-lsb-first-p**).

drawable — The destination **drawable**.

gcontext — The graphics context used to display the image.

data — A sequence of integers.

:start — The index of the first element of *data* displayed.

:depth — The number of bits per pixel displayed. This argument is required.

:x, **:y** — The position in the *drawable* where the image region is displayed. These arguments are required.

:width, **:height — card16** values defining the size of the image region displayed. These arguments are required.

:left-pad — A **card8** specifying the number of leading bits to discard for each image scanline.

:format — One of **:bitmap**, **:xy-pixmap**, or **:z-pixmap**.

FONTS AND CHARACTERS

8

Introduction

8.1 An X server maintains a set of fonts used in the text operations requested by client programs. An X font is an array of character bit maps (or *glyphs*) indexed by integer codes. In fact, font glyphs can also represent cursor shapes or other images and are not limited to character images. X supports both linear and matrix encoding of font indexes. With linear encoding, a font index is interpreted as a single 16-bit integer index into a one-dimensional array of glyphs. With matrix encoding, a font index is interpreted as a pair of 8-bit integer indexes into a two-dimensional array of glyphs. The type of index encoding used is font-dependent.

In order to access or use a font, a client program must first open it using the **open-font** function, sending a font name string as an identifier. **open-font** creates a CLX **font** object used to refer to the font in subsequent functions. Afterward, calling **open-font** with the same font name returns the same **font** object. When a font is no longer in use, a client program can call **close-font** to destroy the **font** object.

A font has several attributes which describe its geometry and its glyphs. CLX provides functions to return the attributes of a font, as well functions for accessing the attributes of individual font glyphs. Glyph attributes are referred to as *character attributes*, since characters are the most common type of font glyphs. A font also has a property list of values recorded by the X server. However, the set of possible font properties and their values are not standardized and are implementation-dependent. Typically, CLX maintains a cache of font and character attributes, in order to minimize server requests. However, the font cache mechanism is implementation-dependent and cannot be controlled by the client. In some cases, CLX may create a *pseudo-font* object solely for the purpose of accessing font attributes. A pseudo-font is represented by a special type of **font** object that cannot be used in a **gcontext**. If necessary, CLX can automatically convert a pseudo-font into a true font, if the name of the pseudo-font is known.

The set of available fonts is server-dependent; that is, font names are not guaranteed to be portable from one server to the next. However, the public X implementation from MIT includes a set of fonts that are typically available with most X servers.

The following paragraphs describe CLX functions to:

- Open and close fonts.
- List available fonts.
- Access font attributes.
- Access character attributes.
- Return the size of a text string.

Opening Fonts

8.2 The following paragraphs discuss the CLX functions for opening and closing fonts.

open-font *display name* Function

Returns:
font — Type **font**.

Opens the font with the given *name* and returns a **font** object. The name string should contain only ISO Latin-1 characters; case is not significant.

display — A **display** object.

name — A font name string.

close-font *font* Function

Deletes the association between the resource ID and the *font*. The *font* is freed when no other server resource references it. The *font* can be unloaded by the X server if this is the last reference to the *font* by any client. In any case, the *font* should never again be referenced because its resource ID is destroyed. This might not generate a protocol request if the *font* is reference-counted locally or if it is a pseudo-font.

font — A **font** object.

discard-font-info *fonts* Function

Discards any state that can be re-obtained with **open-font**. This is simply a performance hint for memory-limited systems.

font — A **font** object.

Listing Fonts

8.3 The following paragraphs describe CLX functions that return fonts or font names that match a given pattern string. Such pattern strings should contain only ISO Latin-1 characters; case is not significant. The following pattern characters can be used for *wildcard* matching:

#* — Matches any sequence of zero or more characters.

#\? — Matches any single character.

For example, the pattern "T?mes Roman" matches the name "Times Roman" but not the name "Thames Roman". However, the pattern "T*mes Roman" matches both names.

font-path *display* &key (**:result-type 'list**) Function

Returns:
paths — Type **sequence** of either **string** or **pathname**.

Returns a **list** (by default) of names containing the current search path for fonts. With **setf**, this function sets the search path for font lookup. There is only one search path per server, not one per client. The interpretation of the names is server-dependent, but they are intended to specify directories to be searched in the order listed.

Setting the path to the empty list restores the default path defined for the server. Note that as a side-effect of executing this request, the server is guaranteed to flush all cached information about fonts for which there are currently no explicit resource IDs allocated.

display — A **display** object.

:result-type — Specifies the type of resulting sequence.

list-font-names *display pattern* &key (**:max-fonts** 65535) (**:result-type 'list**) Function

Returns:
font-name — Type **sequence** of **string**.

Returns a sequence of strings containing the font names that match the *pattern*. The fonts available are determined by the font search path; see **font-path**). The maximum number of font names returned is determined by **:max-fonts**.

display — A **display** object.

pattern — A string used to match font names. Only font names that match the pattern are returned.

:max-fonts — The maximum number of font names returned. Default is 65535.

:result-type — The type of sequence to return. Default is '**list**.

list-fonts *display pattern* &key (**:max-fonts** 65535) (**:result-type 'list**) Function

Returns:
font — Type **sequence** of **font**.

Returns a sequence of pseudo-fonts corresponding to the available fonts whose names match the *pattern*. The fonts available are determined by the font search path; see **font-path**). The maximum number of **font** objects returned is determined by **:max-fonts**.

display — A **display** object.

pattern — A string used to match font names. Only fonts whose name matches the pattern are returned.

:max-fonts — The maximum number of fonts returned. Default is 65535.

:result-type — The type of sequence to return. Default is **'list**.

Font Attributes

8.4 The following paragraphs describe the CLX functions used to access font attributes.

font-all-chars-exist-p *font* Function

Returns:
exists-p — Type **boolean**.

Returns true if glyphs exist for all indexes in the range returned by **font-min-char** and **font-max-char**. Returns **nil** if an index in the range corresponds to empty glyph.

font — A **font** object.

font-ascent *font* Function

Returns:
ascent — Type **int16**.

Returns the vertical *ascent* of the *font* used for interline spacing. The *ascent* defines the nominal distance in pixels from the baseline to the bottom of the previous line of text. Some font glyphs may actually extend beyond the font *ascent*.

font — A **font** object.

font-default-char *font* Function

Returns:
index — Type **card16**.

Returns the *index* of the glyph drawn when an invalid or empty glyph index is specified. If the default index specifies an invalid or empty glyph, an invalid or empty index has no effect.

font — A **font** object.

font-descent *font* Function

Returns:
descent — Type **int16**.

Returns the vertical *descent* of the *font* used for interline spacing. The *descent* defines the nominal distance in pixels from the baseline to the top of the next line of text. Some font glyphs may actually extend beyond the font *descent*.

font — A **font** object.

font-direction *font* Function

Returns:
direction — Type **draw-direction**.

Returns the nominal drawing *direction* for the *font*. The font drawing direction is only a hint that indicates whether the *char-width* of most font glyphs is positive (**:left-to-right** direction) or negative (**:right-to-left** direction). Note that X does not provide any direct support for vertical text.

font — A **font** object.

font-display *font* Function

Returns:
display — Type **display**.

Returns the **display** object associated with the specified *font*.

font — A **font** object.

font-equal *font-1 font-2* Function

Returns true if the two arguments refer to the same server resource and **nil** if they do not.

font-1, *font-2* — The **font** objects.

font-id *font* Function

Returns:
id — Type **resource-id**.

Returns the unique resource ID assigned to the specified *font*.

font — A **font** object.

font-max-byte1 *font* Function

Returns:
max-byte1 — Type **card8**.

Returns zero if the *font* uses linear index encoding. Otherwise, if the *font* uses matrix index encoding, a value between 1 and 255 is returned that specifies the maximum value for the most significant byte of font indexes.

font — A **font** object.

font-max-byte2 *font* Function

Returns:
max-byte2 — Type **card8**.

Returns zero if the *font* uses linear index encoding. Otherwise, if the *font* uses matrix index encoding, a value between 1 and 255 is returned that specifies the maximum value for the least significant byte of font indexes.

font — A **font** object.

font-max-char *font* Function

Returns:
index — Type **card16**.

Returns the maximum valid value used for linear encoded indexes. This function is not meaningful for fonts that use matrix index encoding.

font — A **font** object.

font-min-byte1 *font* Function

Returns:
min-byte1 — Type **card8**.

Returns zero if the *font* uses linear index encoding. Otherwise, if the *font* uses matrix index encoding, a value between 1 and 255 is returned that specifies the minimum value for the most significant byte of font indexes.

font — A **font** object.

font-min-byte2 *font* Function

Returns:
min-byte2 — Type **card8**.

Returns zero if the *font* uses linear index encoding. Otherwise, if the *font* uses matrix index encoding, a value between 1 and 255 is returned that specifies the minimum value for the least significant byte of font indexes.

font — A **font** object.

font-min-char *font* Function

Returns:
index — Type **card16**.

Returns the minimum valid value used for linear encoded indexes. This function is not meaningful for fonts that use matrix index encoding.

font — A **font** object.

font-name *font* Function

Returns:
name — Type **string** or **null**.

Returns the name of the *font*, or **nil** if *font* is a pseudo-font.

font — A **font** object.

font-p *font* Function

Returns:
font-p — Type **boolean**.

Returns true if the argument is a **font** object and **nil** otherwise.

font-plist *font* Function

Returns:
plist — Type **list**.

Returns and (with **setf**) sets the property list for the specified *font*. This function provides a hook where extensions can add data.

font — A **font** object.

font-properties *font* Function

Returns:
properties — Type **list**.

Returns the list of font *properties* recorded by the X server. The returned list is a property list of keyword/value pairs. The set of possible font property keywords is implementation-dependent.

font — A **font** object.

font-property *font name* Function

Returns:
property — Type **int32** or **null**.

Returns the value of the font *property* specified by the *name* keyword. The property value, if it exists, is returned as an uninterpreted 32-bit integer.

font — A **font** object.

name — A font property keyword.

max-char-ascent *font* Function

Returns:
ascent — Type **int16**.

Returns the maximum **char-ascent** value for all characters in *font*.

font — A **font** object.

max-char-attributes *font* Function

Returns:
attributes — Type **int16**.

Returns the maximum **char-attributes** value for all characters in *font*.

font — A **font** object.

max-char-descent *font* Function

Returns:
descent — Type **int16**.

Returns the maximum **char-descent** value for all characters in *font*.

font — A **font** object.

max-char-left-bearing *font* Function

Returns:
left-bearing — Type **int16**.

Returns the maximum **char-left-bearing** value for all characters in *font*.

font — A **font** object.

max-char-right-bearing *font* Function

Returns:
right-bearing — Type **int16**.

Returns the maximum **char-right-bearing** value for all characters in *font*.

font — A **font** object.

max-char-width *font* Function

Returns:
width — Type **int16**.

Returns the maximum **char-width** value for all characters in *font*.

font — A **font** object.

min-char-ascent *font* Function

Returns:
ascent — Type **int16**.

Returns the minimum **char-ascent** for all characters in *font*.

font — A **font** object.

min-char-attributes *font* Function

Returns:
attributes — Type **int16**.

Returns the minimum **char-attributes** for all characters in *font*.

font — A **font** object.

min-char-descent *font* Function

Returns:
descent — Type **int16**.

Returns the minimum **char-descent** for all characters in *font*.

font — A **font** object.

min-char-left-bearing *font* Function

Returns:
left-bearing — Type **int16**.

Returns the minimum **char-left-bearing** for all characters in *font*.

font — A **font** object.

min-char-right-bearing *font* Function

Returns:
right-bearing — Type **int16**.

Returns the minimum **char-right-bearing** for all characters in *font*.

font — A **font** object.

min-char-width *font* Function

Returns:
width — Type **int16**.

Returns the minimum **char-width** for all characters in *font*.

font — A **font** object.

Character Attributes

8.5 The following paragraphs describe the CLX functions used to access the attributes of individual font glyphs.

char-ascent *font index* Function

Returns:
ascent — Type **int16** or **null**.

Returns the vertical distance in pixels from the baseline to the top of the given font glyph. Returns **nil** if the index is invalid or specifies an empty glyph, or if the *font* is a pseudo-font.

font — A **font** object.

index — An **int16** font index.

char-attributes *font index* Function

Returns:
attributes — Type **int16** or **null**.

Returns font-specific *attributes* of the given glyph. The interpretation of such attributes is server-dependent. Returns **nil** if the *index* is invalid or specifies an empty glyph, or if the *font* is a pseudo-font.

font — A **font** object.

index — An **int16** font index.

char-descent *font index* Function

Returns:
descent — Type **int16** or **null**.

Returns the vertical distance in pixels from the baseline to the bottom of the given font glyph. Returns **nil** if the *index* is invalid or specifies an empty glyph, or if the *font* is a pseudo-font.

font — A **font** object.

index — An **int16** font index.

char-left-bearing *font index* Function

Returns:
left-bearing — Type **int16** or **null**.

Returns the left side bearing of the given font glyph. If **draw-glyph** is called with horizontal position *x*, the leftmost pixel of the glyph is drawn at the position (+ *x left-bearing*). Returns **nil** if the *index* is invalid or specifies an empty glyph, or if the *font* is a pseudo-font.

font — A **font** object.

index — An **int16** font index.

char-right-bearing *font index* Function

Returns:
right-bearing — Type **int16** or **null**.

Returns the right side bearing of the given font glyph. If **draw-glyph** is called with horizontal position *x*, the rightmost pixel of the glyph is drawn at the position (+ *x right-bearing*). Returns **nil** if the *index* is invalid or specifies an empty glyph, or if the *font* is a pseudo-font.

font — A *font* object.

index — An **int16** font index.

char-width *font index* Function

Returns:
width — Type **int16** or **null**.

Returns the *width* of the given font glyph. The *width* is defined to be equal to (– *right-bearing left-bearing*). Returns **nil** if the *index* is invalid or specifies an empty glyph, or if the *font* is a pseudo-font.

font — A **font** object.

index — An **int16** font index.

Querying Text Size

8.6 CLX defines functions to return the size of text drawn in a specified font. See paragraph 6.7, Drawing Text, for a description of the **:translate** function used by the functions in the following paragraphs.

text-extents *font sequence* &key (**:start** 0) **:end :translate** Function

Returns:
width — Type **int32**.
ascent — Type **int16**.
descent — Type **int16**.
left — Type **int32**.
right — Type **int32**.
font-ascent — Type **int16**.
direction — Type **draw-direction**.
first-not-done — Type **array-index** or **null**.

Returns the complete geometry of the given *sequence* when drawn in the given *fon*t. The *font* can be a **gcontext**, in which case the font attribute of the given graphics context is used. **:start** and **:end** define the elements of the *sequence* which are used.

The returned *width* is the total pixel width of the translated character sequence. The returned *ascent* and *descent* give the vertical ascent and descent for characters in the translated *sequence*. The returned *left* gives the left bearing of the leftmost character. The returned *right* gives the right bearing of the rightmost character. The returned *font-ascent* and *font-descent* give the maximum vertical ascent and descent for all characters in the *fon*t. If **:translate** causes font changes, then *font-ascent* and *font-descent* will be the maximums over all fonts used. The *direction* returns the preferred draw direction for the font. If **:translate** causes font changes, then the *direction* will be **nil**. The *first-not-done* value returned is **nil** if all elements of the *sequence* were successfully translated; otherwise the index of the first untranslated element is returned.

font — The font (or **gcontext**) used for measuring characters.

sequence — A sequence of characters or other objects to be translated into font indexes.

:start, **:end** — Start and end indexes defining the elements to draw.

:translate — A function to translate text to font indexes. Default is **#'translate-default**.

text-width *font sequence* &key (**:start** 0) **:end :translate** Function

Returns:
width — Type **int32**.
first-not-done — Type **array-index** or **null**.

Returns the total pixel width of the given *sequence* when drawn in the given *font*. The *font* can be a **gcontext**, in which case the font attribute of the given graphics context is used. **:start** and **:end** define the elements of the *sequence* which are used. The second value returned is **nil** if all elements of the *sequence* were successfully translated; otherwise the index of the first untranslated element is returned.

font — The font (or **gcontext**) used for measuring characters.

sequence — A sequence of characters or other objects to be translated into font indexes.

:start, **:end** — Start and end indexes defining the elements to draw.

:translate — A function to translate text to font indexes. Default is **#'translate-default**.

9 COLORS

Colormaps and Colors

9.1 In X, a *color* is defined by a set of three numeric values, representing intensities of red, green, and blue. Red, green, and blue are referred to as the *primary* hues. A *colormap* is a list of colors, each indexed by an integer *pixel* value. Each entry in a colormap is called a color *cell*. Raster graphics displays store pixel values in a special screen hardware memory. As the screen hardware scans this memory, it reads each pixel value, looks up the color in the corresponding cell of a colormap, and displays the color on its screen.

The colormap abstraction applies to all classes of visual types supported by X, including those for screens which are actually monochrome. For example, **:gray-scale** screens use colormaps in which colors actually specify the monochrome intensity. A typical black-and-white monochrome display has a **:static-gray** screen with a two-cell colormap.

The following list describes how pixel values and colormaps are handled for each visual class.

- **:direct-color** — A pixel value is decomposed into separate red, green, and blue subfields. Each subfield indexes a separate colormap. Entries in all colormaps can be changed.
- **:gray-scale** — A pixel value indexes a single colormap that contains monochrome intensities. Colormap entries can be changed.
- **:pseudo-color** — A pixel value indexes a single colormap that contains color intensities. Colormap entries can be changed.
- **:static-color** — Same as **:pseudo-color**, except that the colormap entries are predefined by the hardware and cannot be changed.
- **:static-gray** — Same as **:gray-scale**, except that the colormap entries are predefined by the hardware and cannot be changed.
- **:true-color** — Same as **:direct-color**, except that the colormap entries are predefined by the hardware and cannot be changed. Typically, each of the red, green, and blue colormaps provides a (near) linear ramp of intensity.

CLX provides functions to create colormaps, access and modify colors and color cells, and install colormaps in screen hardware.

Color Functions

9.2 A color is represented by a CLX color object, in which each of the red, green, and blue values is specified by an **rgb-val** — a floating point number between 0.0 and 1.0. (see paragraph 1.6, Data Types). The value 0.0 represents the minimum intensity, while 1.0 represents the maximum intensity. CLX automatically converts **rgb-val** values into 16-bit integers when sending colors to an X server. The X server, in turn, scales 16-bit color values to match the actual intensity range supported by the screen.

Colors used on **:gray-scale** screens must have the same value for each of red, green, and blue. Only one of these values is used by screen hardware to determine intensity; however, CLX does not define which of red, green, or blue is actually used.

The following paragraphs describe the CLX functions used to create, access, and modify colors.

make-color &key (**:blue** 1.0) (**:green** 1.0) (**:red** 1.0) &allow-other-keys Function

Returns:
color — Type **color**.

Creates, initializes, and returns a new **color** object with the specified values for red, green, and blue.

:blue, **:green**, **:red** — **rgb-val** values that specify the saturation for each primary.

color-blue *color* Function

Returns:
blue-intensity — Type **rgb-val**.

Returns and (with **setf**) sets the value for blue in the *color*.

color — A **color** object.

color-green *color* Function

Returns:
green-intensity — Type **rgb-val**.

Returns and (with **setf**) sets the value for green in the *color*.

color — A **color** object.

color-p *color* Function

Returns:
color-p — Type **boolean**.

Returns non-**nil** if the argument is a **color** object and **nil** otherwise.

color-red *color* Function

Returns:
red-intensity — Type **rgb-val**.

Returns and (with **setf**) sets the value for red in the *color*.

color — A **color** object.

color-rgb *color* Function

Returns:
red, *green*, *blue* — Type **rgb-val**.

Returns the values for red, green, and blue in the *color*.

color — A **color** object.

Colormap Functions

9.3 A colormap is represented in CLX by a **colormap** object. A CLX program can create and manipulate several **colormap** objects. However, the colors contained in a **colormap** are made visible only when the **colormap** is *installed*. Each window is associated with a **colormap** that is used to translate window pixels into colors (see **window-colormap**). However, a window will appear in its true colors only if its associated **colormap** is installed.

The total number of colormaps that can be installed depends on the screen hardware. Most hardware devices allow exactly one **colormap** to be installed at any time. That is, **screen-min-installed-maps** and **screen-max-installed-maps** are both equal to 1. Installing a new **colormap** can cause a previously installed **colormap** to be uninstalled. It is important to remember that the set of installed **colormaps** is a hardware resource shared cooperatively among all client programs connected to an X server.

A CLX program can control the contents of **colormaps** by allocating color cells in one of two ways: read-only or read-write. Allocating a read-only color cell establishes a color value for a specified pixel value that cannot be changed. However, read-only color cells can be shared among all client programs. Read-only allocation is the best strategy for making use of limited **colormap** hardware in a multi-client environment.

Alternatively, allocating a read-write color cell allows a client the exclusive right to set the color value stored in the cell. A cell allocated read-write by one client cannot be allocated by another client, not even as a read-only cell. Note that read-write allocation is not allowed for screens whose visual type belongs to one of the **:static-gray**, **:static-color**, or **:true-color** classes. For screens of these classes, **colormap** cells cannot be modified.

Two entries of the default colormap, typically containing the colors black and white, are automatically allocated read-only. The pixel values for these entries can be returned by the functions **screen-black-pixel** and **screen-white-pixel**. Applications that need only two colors and also need to operate on both monochrome and color screens should always use these pixel values. The names *black* and *white* are intended to reflect relative intensity levels and need not reflect the actual colors displayed for these pixel values.

Each screen has a default **colormap**, which is initially installed. By conventions, clients should allocate only read-only cells from the default **colormap**.

Creating Colormaps

9.3.1 CLX provides functions for creating and freeing new **colormap** objects.

create-colormap *visual window* &optional *alloc-p* — Function

Returns:
colormap — Type **colormap**.

Creates and returns a *colormap* of the specified *visual* type for the screen containing the *window*. The *visual* type must be one of those supported by the screen.

Initial color cell values are undefined for visual types belonging to the **:gray-scale**, **:pseudo-color**, and **:direct-color** classes. Color cell values for visual types belonging to the **:static-gray**, **:static-color**, and **:true-color** classes have initial values defined by the visual type. However, X does not define the set of possible visual types or their initial color cell values.

If *alloc-p* is true, all colormap cells are permanently allocated read-write and cannot be freed by **free-colors**. It is an error for *alloc-p* to be true when the visual type belongs to the **:static-gray**, **:static-color**, or **:true-color** classes.

visual — A **visual** type ID.

window — A **window**.

alloc-p — Specifies whether **colormap** cells are permanently allocated read-write.

copy-colormap-and-free *colormap* Function

Returns:
new-colormap — Type **colormap**.

Creates and returns a new **colormap** by copying, then freeing, allocated cells from the specified *colormap*.

All color cells allocated read-only or read-write in the original **colormap** have the same color values and the same allocation status in the *new-colormap*. The values of unallocated color cells in the *new-colormap* are undefined. After copying, all allocated color cells in the original **colormap** are freed, as if **free-colors** was called. The unallocated cells of the original **colormap** are not affected.

If *alloc-p* was true when the original **colormap** was created, then all color cells of the *new-colormap* are permanently allocated read-write, and all the color cells of the original **colormap** are freed.

colormap — A **colormap**.

free-colormap *colormap* Function

Destroys the *colormap* and frees its server resource. If the *colormap* is installed, it is uninstalled. For any window associated with the *colormap*, the window is assigned a **nil colormap**, and a **:colormap-notify** event is generated. The colors displayed for a window with a **nil colormap** are undefined.

However, this function has no effect if the *colormap* is a screen default **colormap**.

colormap — A **colormap**.

Installing Colormaps

9.3.2 The following paragraphs describe the CLX functions to install and uninstall colormaps and to return the set of installed colormaps.

Initially, the default **colormap** for a screen is installed (but is not in the required list).

install-colormap *colormap* Function

Installs the *colormap.* All windows associated with this *colormap* immediately display with true colors. As a side-effect, additional colormaps might be implicitly uninstalled by the server.

If the specified *colormap* is not already installed, a **:colormap-notify** event is generated on every window associated with this *colormap*. In addition, for every other colormap that is implicitly uninstalled, a **:colormap-notify** event is generated on every associated window.

colormap — A **colormap**.

installed-colormaps *window* &key (**:result-type ’list**) Function

Returns:
colormap — Type **sequence** of **colormap**.

Returns a sequence containing the installed **colormaps** for the screen of the specified *window*. The order of the colormaps is not significant.

window — A **window**.

:result-type — A sub-type of **sequence** that indicates the type of sequence to return.

uninstall-colormap *colormap* Function

Uninstalls the *colormap*. However, the *colormap* is not actually uninstalled if this would reduce the set of installed colormaps below the value of **screen-min-installed-maps**. If the *colormap* is actually uninstalled, a **:colormap-notify** event is generated on every associated window.

colormap — A **colormap**.

Allocating Colors

9.3.3 The following paragraphs describe the functions for allocating read-only and read-write color cells, allocating color planes, and freeing color cells.

alloc-color *colormap color* Function

Returns:
pixel — Type **pixel**.
screen-color, *exact-color* — Type **color**.

Returns a *pixel* for a read-only color cell in the *colormap*. The color in the allocated cell is the closest approximation to the requested *color* possible for the screen hardware. The other values returned give both the approximate color stored in the cell and the exact color requested.

The requested *color* can be either a **color** object or a **stringable** containing a color name. If a color name is given, a corresponding color value is looked up (see **lookup-color**) and used. Color name strings must contain only ISO Latin-1 characters; case is not significant.

colormap — A **colormap**.

color — A **color** object or a **stringable** containing a color name.

alloc-color-cells *colormap colors* &key (**:planes** 0) **:contiguous-p** (**:result-type 'list**) Function

Returns:
pixels, mask — Type **sequence** of **pixels**.

Returns a **sequence** of *pixels* for read-write color cells in the *colormap*. The allocated cells contain undefined color values. The visual type class of the **colormap** must be either **:gray-scale**, **:pseudo-color**, or **:direct-color**.

The *colors* argument and the **:planes** argument define the number of pixels and the number of masks returned, respectively. The number of colors must be positive, and the number of planes must be non-negative. A total of (* *colors* (**expt** 2 *planes*)) color cells are allocated. The pixel values for the allocated cells can be computed by combining the returned pixels and masks.

The length of the returned masks sequence is equal to **:planes**. Each mask of the returned masks sequence defines a single bitplane. None of the masks have any 1 bits in common. Thus, by selectively combining masks with **logior**, (**expt** 2 *planes*) distinct combined plane masks can be computed.

The length of the returned *pixels* sequence is equal to *colors*. None of the pixels have any 1 bits in common with each other or with any of the returned masks. By combining pixels and plane masks with **logior**, (* *colors* (**expt** 2 *planes*)) distinct pixel values can be produced.

If the *colormap* class is **:gray-scale** or **:pseudo-color**, each *mask* will have exactly one bit set. If the **colormap** class is **:direct-color**, each *mask* will have exactly three bits set. If **:contiguous-p** is true, combining all masks with **logior** produces a plane mask with either one set of contiguous bits (for **:gray-scale** and **:pseudo-color**) or three sets of contiguous bits (for **:direct-color**).

colormap — A **colormap**.

colors — A positive number defining the length of the pixels sequence returned.

:planes — A non-negative number defining the length of the masks sequence returned.

:contiguous-p — If true, the masks form contiguous sets of bits.

:result-type — A subtype of **sequence** that indicates the type of sequences returned.

alloc-color-planes *colormap colors* &key (**:reds** 0) (**:greens** 0) (**:blues** 0) **:contiguous-p** (:**result-type 'list**) — Function

Returns:
pixels — Type **sequence** of **pixel**.
red-mask, *green-mask*, *blue-mask* — Type **pixel**.

Returns a **sequence** of *pixels* for read-write color cells in the *colormap*. The allocated cells contain undefined color values. The visual type class of the *colormap* must be either **:gray-scale**, **:pseudo-color**, or **:direct-color**.

The *colors* argument defines the number of pixels returned. The **:reds**, **:greens**, and **:blues** arguments define the number of bits set in the returned red, green, and blue masks, respectively. The number of colors must be positive, and the number of bits for each mask must be non-negative. A total of (* *colors* (**expt** 2 (+ *reds greens blues*))) color cells are allocated. The pixel values for the allocated cells can be computed by combining the returned *pixels* and masks.

Each mask of the returned masks defines a pixel subfield for the corresponding primary. None of the masks have any 1 bits in common. By selectively combining subsets of the red, green, and blue masks with **logior**, (**expt** 2 (+ *reds greens blues*) distinct combined plane masks can be computed.

The length of the returned *pixels* **sequence** is equal to *colors*. None of the pixels have any 1 bits in common with each other or with any of the returned masks. By combining pixels and plane masks with **logior**, (* *colors* (**expt** 2 (+ *reds greens blues*)) distinct pixel values can be produced.

If **:contiguous-p** is true, each of returned masks consists of a set of contiguous bits. If the **colormap** class is **:direct-color**, each returned mask lies within the pixel subfield for its primary.

colormap — A **colormap**.

colors — A positive number defining the length of the pixels sequence returned.

:planes — A non-negative number defining the length of the masks sequence returned.

:contiguous-p — If true, then the masks form contiguous sets of bits.

:result-type — A subtype of **sequence** that indicates the type of sequences returned.

free-colors *colormap pixels* &optional (*plane-mask* 0) — Function

Frees a set of allocated color cells from the *colormap*. The pixel values for the freed cells are computed by combining the given *pixels* sequence and **:plane-mask**. The total number of cells freed is:

(* (**length** *pixels*) (**expt** 2 (**logcount** *plane-mask*)))

The **:plane-mask** must not have any bits in common with any of the given *pixels*. The pixel values for the freed cells are produced by using **logior** to combine each of the given pixels with all subsets of the **:plane-mask**.

Note that freeing an individual pixel allocated by **alloc-color-planes** may not allow it to be reused until all related pixels computed from the same plane mask are also freed.

A single error is generated if any computed pixel is invalid or if its color cell is not allocated by the client. Even if an error is generated, all valid pixel values are freed.

colormap — A **colormap**.

pixels — A **sequence** of pixel values.

plane-mask — A pixel value with no bits in common with any of the *pixels*.

Finding Colors

9.3.4 A CLX program can ask the X server to return the colors stored in allocated color cells. The server also maintains a dictionary of color names and their associated color values. CLX provides a function to look up the values for common colors by names such as "red", "purple", and so forth. The following paragraphs describe the CLX functions for returning the color values associated with color cells or with color names.

lookup-color *colormap name* — Function

Returns:
screen-color, *exact-color* — Type **color**.

Returns the color associated by the X server with the given color *name*. The *name* must contain only ISO Latin-1 characters; case is not significant. The first value returned is the closest approximation to the requested color possible on the screen hardware. The second value returned is the true color value for the requested color.

colormap — A **colormap**.

name — A **stringable** color name.

query-colors *colormap pixels* &key (**:result-type 'list**) — Function

Returns:
colors — Type **sequence** of **color**.

Returns a **sequence** of the colors contained in the allocated cells of the *colormap* specified by the given *pixels*. The values returned for unallocated cells are undefined.

colormap — A **colormap**.

pixels — A **sequence** of **pixel** values.

:result-type — A subtype of **sequence** that indicates the type of sequences returned.

Changing Colors

9.3.5 The following paragraphs describe the CLX functions to change the colors in colormap cells.

store-color *colormap pixel color* &key (**:red-p t**) (**:green-p t**) (**:blue-p t**) — Function

Changes the contents of the *colormap* cell indexed by the *pixel*. Components of the given *color* are stored in the cell. The **:red-p**, **:green-p**, and **:blue-p** arguments indicate which components of the given *color* are stored.

The *color* can be either a **color** object or a **stringable** containing a color name. If a color name is given, a corresponding color value is looked up (see **lookup-color**) and used. Color name strings must contain only ISO Latin-1 characters; case is not significant.

colormap — A **colormap**.

pixel — A **pixel**.

color — A color **object** or a **stringable** containing a color name.

:red-p, **:green-p**, **:blue-p** — **boolean** values indicating which color components to store.

store-colors *colormap pixel-colors* &key (**:red-p t**) (**:green-p t**) (**:blue-p t**) Function

Changes the contents of multiple *colormap* cells. *pixel-colors* is a list of the form ({*pixel color*}*), indicating a set of pixel values and the colors to store in the corresponding cells. The **:red-p**, **:green-p**, and **:blue-p** arguments indicate which components of the given colors are stored.

Each color can be either a **color** object or a **stringable** containing a color name. If a color name is given, a corresponding color value is looked up (see **lookup-color**) and used. Color name strings must contain only ISO Latin-1 characters; case is not significant.

colormap — A **colormap**.

pixel-colors — A list of the form ({*pixel color*}*).

:red-p, **:green-p**, **:blue-p** — **boolean** values indicating which color components to store.

Colormap Attributes

9.3.6 The complete set of colormap attributes is discussed in the following paragraphs.

colormap-display *colormap* Function

Returns:
display — Type **display**.

Returns the **display** object associated with the specified *colormap*.

colormap — A **colormap**.

colormap-equal *colormap-1 colormap-2* Function

Returns true if the two arguments refer to the same server resource and **nil** if they do not.

colormap-1, *colormap-2* — A **colormap**.

colormap-id *colormap* Function

Returns:
id — Type **resource-id**.

Returns the unique ID assigned to the specified *colormap*.

colormap — A **colormap**.

colormap-p *colormap* Function

Returns:
map-p — Type **boolean**.

Returns non-**nil** if the argument is a **colormap** and **nil** otherwise.

colormap-plist *colormap* Function

Returns:
colormap-p — Type **boolean**.

Returns and (with **setf**) sets the property list for the specified *colormap*. This function provides a hook where extensions can add data.

colormap — A **colormap**.

CURSORS 10

Introduction

10.1 A *cursor* is a visible shape that appears at the current position of the pointer device. The cursor shape moves with the pointer to provide continuous feedback to the user about the current location of the pointer. Each window can have a cursor attribute that defines the appearance of the pointer cursor when the pointer position lies within the window. See **window-cursor**.

A cursor image is composed of a source bitmap, a mask bitmap, a *hot spot*, a foreground color, and a background color. Either 1-bit pixmaps or font glyphs can be used to specify source and mask bitmaps. The source bitmap identifies the foreground and background pixels of the cursor image; the mask bitmap identifies which source pixels are actually drawn. The mask bitmap thus allows a cursor to assume any shape. The hot spot defines the position within the cursor image that is displayed at the pointer position.

In CLX, a cursor is represented by a **cursor** object. This section describes the CLX functions to:

- Create and free cursor objects
- Change cursor colors
- Inquire the best cursor size
- Access cursor attributes

Creating Cursors

10.2 The following paragraphs describe the CLX functions used to create and free **cursor** objects.

create-cursor &key **:source :mask :x :y :foreground :background** Function

Returns:
cursor — Type **cursor**.

Creates and returns a cursor. **:x** and **:y** define the position of the hot spot relative to the origin of the **:source. :foreground** and **:background** colors must be specified, even if the server only has a **:static-gray** or **:gray-scale** screen. The **:source**, **:x**, and **:y** arguments must also be specified.

The cursor image is drawn by drawing a pixel from the **:source** bitmap at every position where the corresponding bit in the **:mask** bitmap is 1. If the corresponding **:source** bit is 1, a pixel is drawn in the **:foreground** color; otherwise, a pixel is drawn in the **:background** color. If the **:mask** is omitted, all **:source** pixels are drawn. If given, the **:mask** must be the same size as the **:source**.

An X server may not be able to support every cursor size. A server is free to modify any component of the cursor to satisfy hardware or software limitations.

The **:source** and **:mask** can be freed immediately after the cursor is created. Subsequent drawing in the **:source** or **:mask** pixmap has an undefined effect on the cursor.

:source — The source pixmap. This argument is required.

:mask — The mask pixmap.

:x, **:y** — The hot spot position in the **:source**. This argument is required.

:foreground — A **color** object specifying the foreground color. This argument is required.

:background — A **color** object specifying the background color. This argument is required.

create-glyph-cursor &key **:source-font :source-char :mask-font** (**:mask-char** 0) **:foreground :background** — Function

Returns:
cursor — Type **cursor**.

Creates and returns a cursor defined by font glyphs. The source bitmap is defined by the **:source-font** and **:source-char**. The mask bitmap is defined by the **:mask-font** and **:mask-char**. It is an error if the **:source-char** and **:mask-char** are not valid indexes for the **:source-font** and **:mask-font**, respectively. The hot spot position is defined by the "character origin" of the source glyph, that is, the position [– *char-left-bearing*, *char-ascent*] relative to the upper left corner of the source glyph bitmap.

Source and mask bits are compared after aligning the character origins of the source and mask glyphs. The source and mask glyphs need not have the same size or character origin position. If the **:mask-font** is omitted, all source pixels are drawn.

An X server may not be able to support every cursor size. A server is free to modify any component of the cursor to satisfy hardware or software limitations.

Either of the **:source-font** or **:mask-font** can be closed after the cursor is created.

:source-font — The source font. This is a required argument.

:source-char — An index specifying a glyph in the source font. This is a required argument.

:mask-font — The mask font.

:mask-char — An index specifying a glyph in the mask font.

:foreground — A **color** object specifying the foreground color. This is a required argument.

:background — A **color** object specifying the background color. This is a required argument.

free-cursor *cursor* — Function

Destroys the **cursor** object. Cursor server resources are freed when no other references remain.

cursor — A **cursor** object.

Cursor Functions

10.3 The following paragraphs describe the CLX functions used to operate on **cursor** objects.

query-best-cursor *width height display* Function

Returns:
width, *height* — Type **card16**.

Returns the cursor size closest to the requested *width* and *height* that is best suited to the display. The *width* and *height* returned define the largest cursor size supported by the X server. Clients should always be prepared to limit cursor sizes to those supported by the server.

display — A **display** object.

width, *height* — The requested cursor size.

recolor-cursor *cursor foreground background* Function

Changes the color of the specified *cursor*. If the cursor is displayed on a screen, the change is visible immediately.

cursor — A **cursor** object.

foreground — A **color** object specifying the new foreground color.

background — A **color** object specifying the new background color.

Cursor Attributes

10.4 The complete set of cursor attributes is discussed in the following paragraphs.

cursor-display *cursor* Function

Returns:
display — Type **display**.

Returns the **display** object associated with the specified *cursor*.

cursor — A **cursor** object.

cursor-equal *cursor-1 cursor-2* Function

Returns true if the two arguments refer to the same server resource and **nil** if they do not.

cursor-1, *cursor-2* — **cursor** objects.

cursor-id *cursor* Function

Returns:
id — Type **resource-id.**

Returns the unique resource ID that has been assigned to the specified *cursor*.

cursor — A **cursor** object.

cursor-p *cursor*Function

Returns:
cursor-p — Type **boolean**.

Returns true if the argument is a **cursor** object and **nil** otherwise.

cursor-plist *cursor* Function

Returns:
plist — A property list.

Returns and (with **setf**) sets the property list for the specified *cursor*. This function provides a hook where extensions can add data.

cursor — A **cursor** object.

ATOMS, PROPERTIES, AND SELECTIONS

11

Atoms

11.1 In X, an *atom* is a unique ID used as the name for certain server resources — properties and selections.

In CLX, an atom is represented by a keyword symbol. For convenience, CLX functions also allow atoms to be specified by strings and non-keyword symbols. **xatom** is a CLX data type that permits either string or symbol values. A string is equivalent to the **xatom** given by (**intern** *string* **'keyword**). A symbol is equivalent to the **xatom** given by (**intern** (**symbol-name** *symbol*) **'keyword**). The symbol name string of an **xatom** must consist only of ISO Latin characters. Note that the case of **xatom** strings is important; the **xatom** "Atom" is not the same as the **xatom** "ATOM".

Certain atoms are already predefined by every X server. Predefined atoms are designed to represent common names that are likely to be useful for many client applications. Note that these atoms are predefined only in the sense of having **xatom** and **card29** values, not in the sense of having required semantics. No interpretation is placed on the meaning or use of an atom by the server. The **xatom** objects predefined by CLX are listed below.

:arc	**:italic_angle**	**:string**
:atom	**:max_space**	**:subscript_x**
:bitmap	**:min_space**	**:subscript_y**
:cap_height	**:norm_space**	**:superscript_x**
:cardinal	**:notice**	**:superscript_y**
:colormap	**:pixmap**	**:underline_position**
:copyright	**:point**	**:underline_thickness**
:cursor	**:point_size**	**:visualid**
:cut_buffer0	**:primary**	**:weight**
:cut_buffer1	**:quad_width**	**:window**
:cut_buffer2	**:rectangle**	**:wm_class**
:cut_buffer3	**:resolution**	**:wm_client_machine**
:cut_buffer4	**:resource_manager**	**:wm_command**
:cut_buffer5	**:rgb_best_map**	**:wm_hints**
:cut_buffer6	**:rgb_blue_map**	**:wm_icon_name**
:cut_buffer7	**:rgb_color_map**	**:wm_icon_size**
:drawable	**:rgb_default_map**	**:wm_name**
:end_space	**:rgb_gray_map**	**:wm_normal_hints**
:family_name	**:rgb_green_map**	**:wm_size_hints**
:font	**:rgb_red_map**	**:wm_transient_for**
:font_name	**:secondary**	**:wm_zoom_hints**
:full_name	**:strikeout_ascent**	**:x_height**
:integer	**:strikeout_descent**	

When creating a new atom, the following conventions should be obeyed in order to minimize the conflict between atom names:

- Symbol names beginning with an underscore should be used for atoms that are private to a particular vendor or organization. An additional prefix should identify the organization.

- Symbol names beginning with two underscores should be used for atoms that are private to a single application or end user.

CLX provides functions to convert between an **xatom** and its corresponding ID integer. The data type of an atom ID is **card29**. The **xatom** representation is usually sufficient for most CLX programs. However, it is occasionally useful to be able to convert an atom ID returned in events or properties into its corresponding **xatom**.

atom-name *display atom-id* Function

Returns:
atom-name — Type **keyword**.

Returns the atom keyword for the *atom-id* on the given *display* server.

display — A **display** object.

atom-id — A **card29**.

find-atom *display atom-name* Function

Returns:
atom-id — Type **card29** or **null**.

Returns the atom ID for the given *atom-name*, if it exists. If no atom of that name exists for the display server, **nil** is returned.

display — A **display** object.

atom-name — An **xatom**.

intern-atom *display atom-name* Function

Returns:
atom-id — Type **card29** or **null**.

Creates an atom with the given name and returns its atom ID. The atom can survive the interning client; it exists until the last server connection has been closed and the server resets itself.

display — A **display** object.

atom-name — An **xatom**.

Properties

11.2 For each window, an X server can record a set of *properties*. Properties are a general mechanism for clients to associate arbitrary data with a window, and for clients to communicate window data to each other via the server. No interpretation is placed on property data by the server itself.

A property consists of a name, a type, a data format, and data. The name of a property is given by an atom. The property type is another atom used to denote the intended interpretation of the property data. The property formats specifies whether the property data should be treated as a set of 8-, 16-, or 32-bit elements. The property format must be specified so that the X server can communicate property data with the correct byte order.

CLX provides functions to:

- Create or change a property
- Return property data
- List window properties
- Delete a property

change-property *window property data type format* &key (**:mode :replace**) (**:start** 0) **:end :transform** — Function

Creates a new window property or changes an existing property. A **:property-notify** event is generated for the *window.*

If the **:mode** is **:replace**, the new *data*, *type*, and *format* replace any previous values. The subsequence of previous data elements that are replaced is defined by the **:start** and **:end** indexes.

If the **:mode** is **:prepend** or **:append**, no previous data is changed, but the new *data* is added at the beginning or the end, respectively. For these modes, if the *property* already exists, the new *type* and *format* must match the previous values.

The **:transform**, if given, is a function used to compute the actual property data stored. The **:transform**, which must accept a single data element and return a single transformed data element, is called for each data element. If the *data* is a string, the default **:transform** function transforms each character into its ASCII code; otherwise, the default is to store the *data* unchanged.

window — A **window**.

property — A property name **xatom**.

data — A sequence of property data elements.

type — The property type **xatom**.

format — One of 8, 16, or 32.

:mode — One of **:replace**, **:append**, or **:prepend**.

:start, :end — Specify the subsequence of previous data replaced when **:mode** is **:replace**.

:transform — A function that transforms each data element into a data value to store.

delete-property *window property* — Function

Deletes the *window property.* If the *property* already exists, a **:property-notify** event is generated for the *window.*

window — A **window**.

property — A property name **xatom**.

get-property *window property* &key **:type** (**:start** 0) **:end** **:delete-p** (**:result-type** **'list**) **:transform** — Function

Returns:
data — Type **sequence**.
type — Type **xatom**.
format — Type (**member 8 16 32**).
bytes-after — Type **card32**.

Returns a subsequence of the data for the window property. The **:start** and **:end** indexes specify the property *data* elements returned. The **:transform** function is called for elements of the specified subsequence to compute the *data* sequence returned. The property *type* and *format* are also returned. The final return value gives the actual number of data bytes (not elements) following the last data element returned.

If the *property* does not exist, the returned *data* and *type* are **nil** and the returned *format* and *bytes-after* are zero.

If the given **:type** is non-**nil** but does not match the actual property type, then the *data* returned is **nil**, the *type* and *format* returned give the actual property values, and the *bytes-after* returned gives the total number of bytes (not elements) in the property data.

If the given **:type** is **nil** or if it matches the actual property type, then:

- The *data* returned is the transformed subsequence of the property data.
- The *type* and *format* returned give the actual property values.
- The *bytes-after* returned gives the actual number of data bytes (not elements) following the last data element returned.

In this case, the **:delete-p** argument is also examined. If **:delete-p** is true and *bytes-after* is zero, the property is deleted and a **:property-notify** event is generated for the *window*.

window — A **window**.

property — A property name **xatom**.

:type — The requested type **xatom** or **nil**.

:start, **:end** — Specify the subsequence of property *data* returned.

:transform — A function that transforms each data element into a data value to return.

:delete-p — If true, the existing *property* can be deleted.

:result-type — The t*ype* of data sequence to return. Default is **'list**.

list-properties *window* &key (**:result-type 'list**) — Function

Returns:
properties — Type **sequence** of **keyword**.

Returns a sequence containing the names of all *window properties*.

window — A **window**.

:result-type — The type of sequence to return. Default is **'list**.

rotate-properties *window properties* &optional (*delta* 1) — Function

Rotates the values of the given *window properties*. The value of property *i* in the given sequence is changed to the value of the property at index (**mod** (+ *i delta*) (**length** *properties*)). This function operates much like the **rotatef** macro in Common Lisp.

If (**mod** *delta* (**length** *properties*)) is non-zero, a **:property-notify** event is generated on the window for each property, in the same order as they appear in the *properties* sequence.

window — A **window**.

properties — A sequence of **xatom** values.

delta — The index interval between source and destination elements of *properties*.

Selections

11.3 A selection is an atom used to identify data that can be shared among all client programs connected to an X server. Unlike properties, the data represented by a selection is stored by some client program, not by the server.

The data named by a selection is associated with a client window, which is referred to as the *selection owner*. The server always knows which window is the owner of a selection. Selections can be created freely by clients using **intern-atom** to create an atom. CLX provides functions to inquire or change the owner of a selection and to *convert* a selection.

Conversion is the key to the use of selections for inter-client communication. Suppose Client A wants to paste the contents of the data named by selection *S* into his window *WA*. Client A calls **convert-selection** on selection atom *S*, sending a conversion request to the server. The server, in turn, sends a **:selection-request** event to the current owner of *S*, which is window *WB* belonging to Client B. The **:selection-request** event contains the *requestor* window (*WA*), the selection atom (*S*), an atom identifying a requested data type, and the name of a property of *WA* into which the value of *S* will be stored.

Since *WB* is the owner of *S*, it must be associated with the data defined by Client B as the value of *S*. When *WB* gets the **:selection-request** event, Client B is expected to convert the value of *S* to the requested data type (if possible) and store the converted value in the given requestor property. Client B is then expected to send a **:selection-notify** event to the requestor window *WA*, informing the requestor that the converted value for *S* is ready. Upon receiving the **:selection-notify** event, Client A can call **get-property** to retrieve the converted value and to paste it into *WA*.

NOTE: Clients using selections must always be prepared to handle **:selection-request** events and/or **:selection-notify** events. There is no way for a client to ask not to receive these types of events.

Type atoms used in selection conversion can represent arbitrary client-defined interpretations of the selection data. For example, if the value of selection *S* is a text string, Client A might request its typeface by requesting conversion to the **:font** type. A type **atom** can also represent a request to the selection owner to perform some action as a side-effect of conversion (for example, **:delete**). Some of the predefined atoms of an X server are intended to be used as selection types (for example, **:colormap**, **:bitmap**, **:string**, and so forth) However, X does not impose any requirements on the interpretation of type atoms.

When multiple clients negotiate for ownership of a selection, certain race conditions might be possible. For example, two clients might each receive a user command to assert ownership of the **:primary** selection, but the order in which the server processes these client requests is unpredictable. As a result, the ownership request initiated most recently by the user might be incorrectly overridden by the other earlier ownership request. To prevent such anomalies, the server records a *last-changed* timestamp for each change of selection ownership.

Although inter-client communication via selections is rather complex, it offers important benefits. Since selection communication is mediated by an X server, clients can share data even though they are running on different hosts and using different networking protocols. Data storage and conversion is distributed among clients so that the server is not required to provide all possible data types or to store multiple forms of selection data.

Certain predefined atoms are used as standard selections, as described in the X11 Inter-client Communications Conventions Manual. Some of the standard selections covered by these conventions are:

- **:primary** — The *primary selection*. The main vehicle for inter-client cut and paste operations.

- **:secondary** — The *secondary selection*. In some environments, clients can use this as an auxiliary to **:primary**.

- **:clipboard** — Analogous to akill ring. Represents the most recently deleted data item.

convert-selection *selection type requestor* &optional *property time* Function

Requests that the value of the *selection* be converted to the specified *type* and stored in the given *property* of the *requestor* window.

If the *selection* has an owner, the X server sends a :**selection-request** event to the owner window. Otherwise, if no owner exists, the server generates on the requestor a **:selection-notify** event containing a **nil** *property* atom.

The given *property* specifies the requestor property that will receive the converted value. If the *property* is omitted, the *selection* owner will define a property to use. The *time* furnishes a timestamp representing the time of the conversion request; by default, the current server time is used.

NOTE: Standard conventions for inter-client communication require that both the requestor property and the time must be specified. If possible, the time should be the time of a user event which initiated the conversion. Alternatively, a timestamp can be obtained by calling **change-property** to append zero-length data to some property; the timestamp in the resulting **:property-notify** event can then be used.

---

selection — The **xatom** for the selection name.

type — The **xatom** for the requested data type.

requestor — The **window** to receive the converted *selection* value.

property — The **xatom** for the requestor property to receive the converted value.

time — A **timestamp**.

selection-owner *display selection* &optional *time* Function

Returns:
owner — Type **window** or **null**.

Returns and (with **setf**) changes the owner and the last-changed *time* for the *selection*. If the owner is **nil**, no owner for the *selection* exists. When the owner window for a *selection* is destroyed, the *selection* owner is set to **nil** without affecting the last-changed *time*.

The *time* argument is used only when changing the *selection* owner. If the *time* is **nil**, the current server time is used. If the *time* is earlier than the current last-changed time of the *selection* or if the *time* is later than the current server time, the owner is not changed. Therefore, a client should always confirm successful change of ownership by immediately calling **selection-owner**. If the change in ownership is successful, the last-changed time of the *selection* is set to the specified *time*.

If the change in ownership is successful and the new owner is different from the previous owner, and if the previous owner is not **nil**, a **:selection-clear** event is generated for the previous owner window.

NOTE: Standard conventions for inter-client communication require that a non-nil time must be specified. If possible, the time should be the time of a user event which initiated the change of ownership. Alternatively, a timestamp can be obtained by calling change-property to append zero-length data to some property; the timestamp in the resulting **:property-notify** event can then be used.

---

display — A **display**.

selection — The **xatom** for the selection name.

time — A **timestamp**.

12

EVENTS AND INPUT

Introduction

12.1 A client application uses CLX functions to send *requests* to an X server over a display connection returned by the **open-display** function. In return, the X server sends back *replies* and *events*. Replies are synchronized with specific requests and return requested server information. Events typically occur asynchronously. Device events are generated by user input from both the keyboard and pointer devices. Other events are side-effects of the requests sent by CLX functions. The types of events returned by an X server are summarized below.

Device Events	Events Returned
Keyboard	**:key-press** **:key-release**
Pointer	**:button-press** **:button-release** **:enter-notify** **:leave-notify** **:motion-notify**

Side-Effect Events	Events Returned
Client communication	**:client-message** **:property-notify** **:selection-clear** **:selection-notify** **:selection-request**
Color map state	**:colormap-notify**
Exposure	**:exposure** **:graphics-exposure** **:no-exposure**
Input focus	**:focus-in** **:focus-out**
Keyboard and pointer state	**:keymap-notify** **:mapping-notify**
Structure control	**:circulate-request** **:configure-request** **:map-request** **:resize-request**
Window state	**:circulate-notify** **:configure-notify** **:create-notify** **:destroy-notify** **:gravity-notify** **:map-notify** **:reparent-notify** **:unmap-notify** **:visibility-notify**

Client programs can override the server's normal distribution of events by *grabbing* the pointer or the keyboard. Grabbing causes events from the pointer or keyboard device to be reported to a single specified window, rather than to their ordinary destinations. It can also cause the server to *freeze* the grabbed device, sending queued events only when explicitly requested by the grabbing client. Two kinds of grabs are possible:

- Active — Events are immediately grabbed.
- Passive — Events are grabbed later, as soon as a specified device event occurs.

Grabbing an input device is performed rarely and usually only by special clients, such as window managers.

This section describes the CLX functions used to:

- Select events
- Process an event on the event queue
- Manage the event queue
- Send events to other applications
- Read and change the pointer position
- Manage the keyboard input focus
- Grab pointer and keyboard events
- Release queued events

This section also contains a detailed description of the content of each type of event.

Selecting Events

12.2 A client *selects* which types of events it receives from a specific window. The window event-mask attribute, set by the client, determines which event types are selected (see **window-event-mask** in paragraph 4.3, Window Attributes). Most types of events are received by a client only if they are selected for some window.

In the X protocol, an event-mask is represented as a bit string. CLX also allows an event mask to be defined by a list of **event-mask-class** keywords. The functions **make-event-keys** and **make-event-mask** can be used to convert between these two forms of an event-mask. In general, including an **event-mask-class** keyword in an event-mask causes one or more related event types to be selected. The following table describes the event types selected by each **event-mask-class** keyword.

Event Mask Keyword	Event Types Selected
:button-1-motion	**:motion-notify** when **:button-1** is down
:button-2-motion	**:motion-notify** when **:button-2** is down
:button-3-motion	**:motion-notify** when **:button-3** is down
:button-4-motion	**:motion-notify** when **:button-4** is down
:button-5-motion	**:motion-notify** when **:button-5** is down
:button-motion	**:motion-notify** when any pointer button is down
:button-press	**:button-press**
:button-release	**:button-release**
:colormap-change	**:colormap-notify**
:enter-window	**:enter-notify**
:exposure	**:exposure**
:focus-change	**:focus-in, :focus-out**
:key-press	**:key-press**
:key-release	**:key-release**
:keymap-state	**:keymap-notify**
:leave-window	**:leave-notify**
:owner-grab-button	Pointer events while button is grabbed
:pointer-motion	**:motion-notify**
:pointer-motion-hint	Single **:motion-notify** only
:property-change	**:property-notify**
:resize-redirect	**:resize-request**
:structure-notify	**:circulate-notify, :configure-notify, :destroy-notify, : gravity-notify, :map-notify, :reparent-notify, :unmap-notify**
:substructure-redirect	**:circulate-request, :configure-request, :map-request**
:visibility-change	**:visibility-notify**

Some types of events do not have to be selected to be received and therefore are not represented in an event-mask. For example, the **copy-plane** and **copy-area** functions cause **:graphics-exposure** and **:no-exposure** events to be reported, unless exposures are turned **:off** in the graphics context (see **copy-area** and **copy-plane** in paragraph 6.2, Area and Plane Operations, and **gcontext-exposures** in paragraph 5.4.6, Exposures). Also, **:selection-clear**, **:selection-request**, **:selection-notify** and **:client-message** events can be received at any time, but they are generally sent only to clients using selections (see paragraph 12.12.7, Client Communications Events). **:mapping-notify** is always sent to clients when the keyboard mapping is changed.

Any client can select events for any window. A window maintains a separate event-mask for each interested client. In general, multiple clients can select for the same events on a window. After the X server generates an event, it sends it to all clients which selected it. However, the following restrictions apply to sharing window events among multiple clients. For a given window:

- Only one client at a time can include **:substructure-redirect** in its event-mask
- Only one client at a time can can include **:button-press** in its event-mask
- Only one client at a time can include **:resize-redirect** in its event-mask

Processing Events

12.3 Events received by a CLX client are stored in an *event queue* until they are read and processed. Events are processed by *handler functions*.

handler-function &rest *event-slots* &key **:display :event-key :send-event-p** &allow-other-keys — Function

Returns:
handled-p — Type **boolean**.

The arguments to a handler function are keyword-value pairs that describe the contents of an event. The actual *event-slots* passed depend on the event type, except that **:display**, **:event-key**, and **:send-event-p** are given for all event types. The keyword symbols used for each event type are event slot names defined by the **declare-event** macro and are described in paragraph 12.12.8, Declaring Event Types.

If a handler returns non-**nil**, the event is considered *processed* and can be removed from the event queue. Otherwise, if a handler function returns **nil**, the event can remain in the event queue for later processing.

:display — A **display** for the connection that returned the event.

:event-key — An **event-key** keyword specifying the event type.

:send-event-p — If true, the event was sent from another application using the **send-event** function.

process-event *display* &key **:handler :timeout :peek-p :discard-p (:force-output-p t)** — Function

Returns:
handled-p — Type **boolean**.

Invokes **:handler** on each queued event until **:handler** returns non-**nil**. Then, the non-**nil :handler** value is returned by **process-event**. If **:handler** returns **nil** for each event in the event queue, **process-event** waits for another event to arrive. If timeout is non-**nil** and no event arrives within the specified timeout interval (given in seconds), **process-event** returns **nil**; if timeout is **nil**, **process-event** will not return until **:handler** returns non-**nil**. **process-event** may wait only once on network data, and therefore timeout prematurely.

If **:force-output-p** is true, **process-event** first invokes **display-force-output** to send any buffered requests. If **:peek-p** is true, a processed event is not removed from the queue. If **:discard-p** is true, unprocessed events are removed from the queue; otherwise, unprocessed events are left in place.

If **:handler** is a sequence, it is expected to contain handler functions for each event type. The sequence index of the handler function for a particular event type is given by (**position event-key *event-key-vector***).

display — A **display**.

:handler — A handler function or a sequence of handler functions.

:timeout — Specifies the timeout delay in seconds.

:peek-p — If **nil**, events are removed from the event queue after processing.

:discard-p — If true, unprocessed events are discarded.

:force-output-p — If true, buffered output requests are sent.

event-case *display* &key **:timeout :peek-p :discard-p (:force-output-p t)** Macro
&body *clauses*
Returns:
handled-p — Type **boolean**.

Executes the matching clause for each queued event until a clause returns non-**nil**. The non-**nil** clause value is then returned. Each of the clauses is a list of the form (*event-match* [*event-slots*] &rest *forms*), where:

event-match — Either an **event-key**, a list of **event-keys**, otherwise, or **t**. It is an error for the same key to appear in more than one clause.

event-slots — If given, a list of (non-keyword) event slot symbols defined for the specified event type(s). See paragraph 12.12.8, Declaring Event Types.

forms — A list of forms that process the specified event type(s). The value of the last form is the value returned by the clause.

A clause matches an event if the **event-key** is equal to or a member of the *event-match*, or if the *event-match* is **t** or **otherwise**. If no **t** or **otherwise** clause appears, it is equivalent to having a final clause that returns **nil**. If *event-slots* is given, these symbols are bound to the value of the corresponding event slot in the clause forms. Each element of *event-slots* can also be a list of the form (*event-slot-keyword variable*), in which case the *variable* symbol is bound to the value of the event slot specified by the *event-slot-keyword*.

If every clause returns **nil** for each event in the event queue, **event-case** waits for another event to arrive. If **:timeout** is non-**nil** and no event arrives within the specified timeout interval (given in seconds), **event-case** returns **nil**; if **:timeout** is **nil**, **event-case** will not return until a clause returns non-**nil**. **event-case** may wait only once on network data and therefore timeout prematurely.

If **:force-output-p** is true, **event-case** first invokes **display-force-output** to send any buffered requests. If **:peek-p** is true, a processed event is not removed from the queue. If **:discard-p** is true, unprocessed events are removed from the queue; otherwise, unprocessed events are left in place.

display — A **display**.

:handler — A handler function or a sequence of handler functions.

:timeout — Specifies the timeout delay, in seconds.

:peek-p — If **nil**, events are removed from the event queue after processing.

:discard-p — If true, unprocessed events are discarded.

:force-output-p — If true, buffered output requests are sent.

clauses — Code to process specified event types.

event-cond *display* &key **:timeout :peek-p :discard-p (:force-output-p t)** Macro
&body *clauses*
Returns:
handled-p — Type **boolean**.

Similar to **event-case** except that each of the clauses is a list of the form (*event-match* [*event-slots*] *test-form* &rest *forms*). Executes the *test-form* of the clause that matches each queued event until a *test-form* returns non-**nil**. The body *forms* of the clause are then executed. The values returned by the last clause body form are then returned by **event-cond**.

When a *test-form* returns true and **:peek-p** is **nil**, or when a *test-form* returns **nil** and **:discard-p** is true, the matching event is removed from the event queue before the body *forms* are executed.

display — A **display**.

:handler — A handler function or a sequence of handler functions.

:timeout — Specifies the timeout delay in seconds.

:peek-p — If **nil**, events are removed from the event queue after processing.

:discard-p — If true, unprocessed events are discarded.

:force-output-p — If true, buffered output requests are sent.

clauses — Code to process specified event types.

Managing the Event Queue

12.4 The following paragraphs describe CLX functions and macros used to:

- Put a new event on the event queue
- Discard the current event
- Return the current length of the event queue
- Gain exclusive access to the event queue for a client process

queue-event *display event-key* &rest *event-slots* &key **:append-p** &allow-other-keys — Function

Places an event of the type given by *event-key* into the event queue. When **:append-p** is true, the event is placed at the tail of the queue; otherwise, the event is placed at the head of the queue. The actual *event-slots* passed depend on the event type. The keyword symbols used for each event type are event slot names defined by the **declare-event** macro and are described in paragraph 12.12.8, Declaring Event Types.

display — A **display**.

event-key — Specifies the type of event placed in the queue.

event-slots — Keyword-value pairs that describe the contents of an event.

:append-p — If true, the event is placed at the tail of the queue; otherwise, the event is placed at the head of the queue.

discard-current-event *display* — Function

Returns:
discarded-p — Type **boolean**.

Discards the current event for the *display*. Returns **nil** when the event queue is empty; otherwise, returns **t**. This function provides extra flexibility for discarding events, but it should be used carefully; use **event-cond** instead, if possible. Typically, **discard-current-event** is called inside a handler function or a clause of an **event-case** form and is followed by another call to **process-event**, **event-case**, or **event-cond**.

display — A **display**.

event-listen *display* &optional (**timeout** 0) Function

Returns:
event-count — Type **(or null integer)**.

Returns the number of events queued locally. If the event queue is empty, **event-listen** waits for an event to arrive. If timeout is non-**nil** and no event arrives within the specified timeout interval (given in seconds), **event-listen** returns **nil**; if timeout is **nil**, **event-listen** will not return until an event arrives.

display — A **display**.

timeout — The number of seconds to wait for events.

with-event-queue *display* &body *body* Macro

Executes the *body* in a critical region in which the executing client process has exclusive access to the event queue.

display — A **display**.

body — Forms to execute.

Sending Events

12.5 A client can send an event to a window. Clients selecting this window event will receive it just like any other event sent by the X server.

send-event *window event-key event-mask* &rest *event-slots* &key **:propagate-p :display** &allow-other-keys Function

Sends an event specified by the *event-key* and *event-slots* to the given destination *window*. Any active grabs are ignored. The *event-slots* passed depend on the event type. The keyword symbols used for each event type are event slot names defined by the **declare-event** macro and are described in paragraph 12.12.8, Declaring Event Types.

If the *window* is **:pointer-window**, the destination *window* is replaced with the window containing the pointer. If the *window* is **:input-focus**, the destination *window* is replaced with the descendant of the focus window that contains the pointer or (if no such descendant exists) the focus window. The **:display** keyword is only required if the *window* is **:pointer-window** or **:input-focus**.

The *event-key* must be one of the core events, or one of the events defined by an extension, so the server can send the event with the correct byte-order. The contents of the event are otherwise unaltered and unchecked by the server, except that the **send-event-p** event slot is set to true.

If the *event-mask* is **nil**, the event is sent to the client that created the destination *window* with an *event-mask* of 0; if that client no longer exists, no event is sent. Otherwise, the event is sent to every client selecting any of the event types specified by *event-mask* on the destination *window*.

If **:propagate-p** is true and no clients have selected any of the event types in *event-mask* on the destination *window*, the destination is replaced with the closest ancestor of *window* for which some client has selected a type in *event-mask* and no intervening window has that type in its do-not-propagate mask. If no such window exists, or if the *window* is an ancestor of the focus window and **:input-focus** was originally specified as the destination, the event is not sent to any clients. Otherwise, the event is reported to every client selecting on the final destination any of the types specified in *event-mask*.

window — The destination **window** for the event.

event-key — An **event-key** defining the type of event to send.

event-mask — Specifies the event types that receiving clients must select.

event-slots — Keyword-value pairs that describe the contents of an event.

:propagate-p — If true, the event can be propagated to ancestors of the destination window.

:display — A **display**.

Pointer Position

12.6 The CLX functions affecting pointer position are discussed in the following paragraphs.

query-pointer *window* Function

Returns:
x — Type **int16**.
y — Type **int16**.
same-screen-p — Type **boolean**.
child — Type **window** or **null**.
state-mask — Type **card16**.
root-x — Type **int16**.
root-y — Type **int16**.
root — Type **window**.

Returns the current pointer coordinates relative to the given *window*. If **query-pointer** returns **nil** for *same-screen-p*, the pointer is not on the same screen as the *window*. In this case, **query-pointer** returns a value of **nil** for *child* and a value of zero for *x* and *y*. If **query-pointer** returns true for *same-screen-p*, the returned *x* and *y* are relative to the origin of window. The *child* is the child of the window containing the pointer, if any. The *state-mask* returned gives the current state of the modifier keys and pointer buttons. The returned *root* is the root window currently containing the pointer. The returned *root-x* and *root-y* specify the pointer coordinates relative to *root*.

window — A **window** specifying the coordinate system for the returned position.

global-pointer-position *display* Function

Returns:
root-x — Type **int16**.
root-y — Type **int16**.
root — Type **window**.

Returns the *root* window currently containing the *display* pointer and the current position of the pointer relative to the *root*.

display — A **display**.

pointer-position *window* Function

Returns:
x — Type **int16**.
y — Type **int16**.
same-screen-p — Type **boolean**.
child — Type **window** or **null**.

Returns the current pointer coordinates relative to the given *window*. If **pointer-position** returns **nil** for *same-screen-p*, the pointer is not on the same screen as the *window*. In this case, **pointer-position** returns a value of **nil** for *child* and a value of zero for *x* and *y*. If **pointer-position** returns true for *same-screen-p*, the returned *x* and *y* are relative to the origin of *window*.

window — A **window** specifying the coordinate system for the returned position.

motion-events w*indow* &key **:start :stop** (**:result-type 'list**) Function

Returns:
motion-events — Type **(repeat-seq (int16 x) (int16 y) (timestamp time))**.

Many X server implementations maintain a more precise history of pointer motion between event notifications. The pointer position at each pointer hardware interrupt can be stored into a buffer for later retrieval.This is called the *motion history buffer*. A paint program, for example, may want to have a precise history of where the pointer traveled, even though for most other applications this amount of detail is grossly excessive.

The **motion-events** function returns all events in the motion history buffer that fall between the specified **:start** and **:stop** timestamps (inclusive) and have coordinates that lie within the specified *window* (including borders) at its present placement. If the **:start** time is later than the **:stop** time or if the **:start** time is in the future, no events are returned.

window — The **window** containing the returned motion events.

:start, :stop — timestamp values for the time interval for returned motion events.

:result-type — The form of the returned motion events.

warp-pointer *destination destination-x destination-y* Function

Moves the pointer to the given coordinates relative to the *destination* window. **warp-pointer** should be rarely be used since the user should normally be in control of the pointer position. **warp-pointer** generates events just as if the user had instantaneously moved the pointer from one position to another.

warp-pointer cannot move the pointer outside the confine-to window of an active pointer grab; an attempt to do so only moves the pointer as far as the closest edge of the confine-to window.

destination — The **window** into which the pointer is moved.

destination-x, destination-y — The new position of the pointer relative to the destination.

warp-pointer-relative *display x-offset y-offset* Function

Moves the pointer by the given offsets. This function should rarely be used since the user should normally be in control of the pointer position. **warp-pointer-relative** generates events just as if the user had instantaneously moved the pointer from one position to another.

warp-pointer-relative cannot move the pointer outside the confine-to window of an active pointer grab; an attempt to do so only moves the pointer as far as the closest edge of the confine-to window.

display — A **display**.

x-offset, y-offset — The offsets used to adjust the pointer position.

warp-pointer-if-inside *destination destination-x destination-y* Function
source source-x source-y &optional (*source-width* 0)
(*source-height* 0)

Moves the pointer to the given position relative to the *destination* window. However, the move can only take place if the pointer is currently contained in a visible portion of the specified rectangle of the *source* window. If *source-height* is zero, it is replaced with the current height of *source* window minus *source-y*. If *source-width* is zero, it is replaced with the current width of *source* window minus *source-x*.

warp-pointer-if-inside generates events just as if the user had instantaneously moved the pointer from one position to another. **warp-pointer-if-inside** cannot move the pointer outside the confine-to window of an active pointer grab; an attempt to do so only moves the pointer as far as the closest edge of the confine-to window.

destination — The **window** into which the pointer is moved.

destination-x, destination-y — The new position of the pointer relative to the *destination*.

source — The **window** that must currently contain the pointer.

source-x, source-y, source-width, source-height —The source rectangle that must currently contain the pointer.

warp-pointer-relative-if-inside *x-offset y-offset source source-x source-y* &optional (*source-width* 0) (*source-height* 0) Function

Moves the pointer by the given offsets. However, the move can only take place if the pointer is currently contained in a visible portion of the specified rectangle of the *source* window. If *source-height* is zero, it is replaced with the current height of *source-window* minus *source-y*. If *source-width* is zero, it is replaced with the current width of *source-window* minus *source-x*.

warp-pointer-relative-if-inside generates events just as if the user had instantaneously moved the pointer from one position to another. **warp-pointer-relative-if-inside** cannot move the pointer outside the confine-to window of an active pointer grab; an attempt to do so only moves the pointer as far as the closest edge of the confine-to window.

x-offset, y-offset — The offsets used to adjust the pointer position.

source — The **window** that must currently contain the pointer.

source-x, source-y, source-width, source-height — The source rectangle that must currently contain the pointer.

Managing Input Focus

12.7 CLX provides the **set-focus-input** and **focus-input** functions to set and get the keyboard input focus window.

set-input-focus *display focus revert-to* &optional *time* Function

Changes the keyboard input focus and the last-focus-change time. The function has no effect if the specified *time* is earlier than the current last-focus-change time or is later than the current server time; otherwise, the last-focus-change time is set to the specified *time*. The **set-input-focus** function causes the X server to generate **:focus-in** and **:focus-out** events.

If **:none** is specified as the *focus*, all keyboard events are discarded until a new focus window is set. In this case, the *revert-to* argument is ignored.

If a window is specified as the *focus* argument, it becomes the keyboard's focus window. If a generated keyboard event would normally be reported to this window or one of its inferiors, the event is reported normally; otherwise, the event is reported with respect to the focus window.

If **:pointer-root** is specified as the *focus* argument, the input focus window is set to the root window of the screen containing the pointer when each keyboard event occurs. In this case, the *revert-to* argument is ignored.

The specified *focus* window must be viewable at the time of the request. If the *focus* window later becomes not viewable, the new focus window depends on the *revert-to* argument. If *revert-to* is specified as **:parent**, the *focus* reverts to the parent (or the closest viewable ancestor) and the new *revert-to* value is take to be **:none**. If *revert-to* is **:pointer-root** or **:none**, the *focus* reverts to that value. When the *focus* reverts, **:focus-in** and **:focus-out** events are generated, but the last-focus-change time is not affected.

display — A **display**.

focus — The new input focus **window**.

revert-to — The focus **window** when focus is no longer viewable.

time — A **timestamp**.

input-focus *display* Function

Returns:
focus — Type (**or window** (**member :none :pointer-root**)).
revert-to — Type (**or window** (**member :none :pointer-root :parent**)).

Returns the *focus* window, **:pointer-root**, or **:none**, depending on the current state of the focus window. *revert-to* returns the current focus revert-to state.

display — A **display**.

Grabbing the Pointer

12.8 CLX provides the **grab-pointer** and **ungrab-pointer** functions for grabbing and releasing pointer control.

grab-pointer *window event-mask* &key **:owner-p :sync-pointer-p :sync-keyboard-p :confine-to :cursor :time** Function

Returns:
grab-status — One of **:already-grabbed**, **:frozen**, **:invalid-time**, **:not-viewable**, or **:success**.

Actively grabs control of the pointer. Further pointer events are only reported to the grabbing client. The request overrides any active pointer grab by this client.

If **:owner-p** is **nil**, all generated pointer events are reported with respect to *window*, and are only reported if selected by *event-mask*. If **:owner-p** is true, and if a generated pointer event would normally be reported to this client, it is reported normally; otherwise the event is reported with respect to the *window*, and is only reported if selected by *event-mask*. For either value of **:owner-p**, unreported events are simply discarded.

If **:sync-pointer-p** is **nil**, pointer event processing continues normally (asynchronously); if the pointer is currently frozen by this client, then processing of pointer events is resumed. If **:sync-pointer-p** is true (indicating a synchronous action), the pointer (as seen via the protocol) appears to freeze, and no further pointer events are generated by the server until the grabbing client issues a releasing **allow-events** request. Actual pointer changes are not lost while the pointer is frozen; they are simply queued for later processing.

If **:sync-keyboard-p** is **nil**, keyboard event processing is unaffected by activation of the grab. If **:sync-keyboard-p** is true, the keyboard (as seen via the protocol) appears to freeze, and no further keyboard events are generated by the server until the grabbing client issues a releasing **allow-events** request. Actual keyboard changes are not lost while the keyboard is frozen; they are simply queued for later processing.

If **:cursor** is specified, it is displayed regardless of what window the pointer is in. Otherwise, the normal cursor for the *window* is displayed.

If a **:confine-to** window is specified, the pointer is restricted to stay within that window. The :**confine-to** window does not need to have any relationship to the *window*. If the pointer is not initially in the **:confine-to** window, it is warped automatically to the closest edge (with **:enter/:leave-events** generated normally) just before the grab activates. If the **:confine-to** window is subsequently reconfigured, the pointer is warped automatically as necessary to keep it contained in the window.

grab-pointer generates **:enter-notify** and **:leave-notify** events. **grab-pointer** can fail with a status of:

- **:already-grabbed** if the pointer is actively grabbed by some other client
- **:frozen** if the pointer is frozen by an active grab of another client
- **:not-viewable** if the *window* or the **:confine-to** window is not viewable, or if the **:confine-to** window lies completely outside the boundaries of the root window.
- **:invalid-time** if the specified time is earlier than the last-pointer-grab time or later than the current server time. Otherwise, the last-pointer-grab time is set to the specified time, with current-time replaced by the current server time, and a value of **:success** is returned by **grab-pointer**.

window — The **window** grabbing the pointer.

event-mask — A **pointer-event-mask**.

:owner-p — If true, all client windows receive pointer events normally.

:sync-pointer-p — Indicates whether the pointer is in synchronous or asynchronous mode.

:sync-keyboard-p — Indicates whether the keyboard is in synchronous or asynchronous mode.

:confine-to — A **window** to which the pointer is confined.

:cursor — A **cursor**.

:time — A **timestamp**. A **nil** value means the current server time is used.

ungrab-pointer *display* &key **:time** Function

Releases the pointer if this client has it actively grabbed (from either **grab-pointer**, **grab-button**, or from a normal button press), and releases any queued events. The request has no effect if the specified **:time** is earlier than the last-pointer-grab time or is later than the current server time. An **ungrabpointer** is performed automatically if the event window or **:confine-to** window for an active pointer grab becomes not viewable.

This request generates **:enter-notify** and **:leave-notify** events.

display — A **display**.

:time — A **timestamp**.

change-active-pointer-grab *display event-mask* &optional *cursor time* Function

Changes the specified dynamic parameters if the pointer is actively grabbed by the client and the specified *time* is no earlier than the last-pointer-grab time and no later than the current server time. The interpretation of *event-mask* and *cursor* are as in **grab-pointer**. **change-active-pointer-grab** has no effect on the passive parameters of a **grab-button**.

display — A **display**.

event-mask — A **pointer-event-mask**.

cursor — A **cursor** or **nil**.

time — A **timestamp**.

Grabbing a Button

12.9 CLX provides the **grab-button** and **ungrab-button** functions for passively grabbing and releasing pointer control.

grab-button *window button event-mask* &key (**:modifiers** 0) **:owner-p** **:sync-pointer-p :sync-keyboard-p :confine-to :cursor** Function

This request establishes a passive grab. If the specified *button* is pressed when the specified modifier keys are down (and no other buttons or modifier keys are down), and:

- *window* contains the pointer
- The **:confine-to** window (if any) is viewable
- These constraints are not satisfied for any ancestor of *window*

then:

- The pointer is actively grabbed as described with **grab-pointer**
- The last-pointer-grab time is set to the time that the button was pressed (as transmitted in the **:button-press** event)
- The **:button-press** event is reported

The interpretation of the remaining arguments is the same as with **grab-pointer**. The active grab is terminated automatically when all buttons are released (independent of the state of modifier keys).

A zero *modifier* mask is equivalent to issuing the request for all possible modifier-key combinations (including the combination of no modifiers). It is not required that all specified modifiers have currently assigned keycodes. A *button* of **:any** is equivalent to issuing the request for all possible buttons. Otherwise, it is not required that the specified *button* currently be assigned to a physical button.

window — A **window**.

button — The button (type **card8**) pressed or **:any**.

event-mask — A **pointer-event-mask**.

:modifiers — A **modifier-mask**.

:owner-p — If true, all client windows receive pointer events normally.

:sync-pointer-p — Indicates whether the pointer is handled in a synchronous or asynchronous fashion.

:sync-keyboard-p — Indicates whether the keyboard is in synchronous or asynchronous mode.

:confine-to — A **window** to which the pointer is confined.

:cursor — A **cursor**.

ungrab-button *window button* &key (**:modifiers** 0) Function

Releases the passive button/key combination on the specified *window* if it was grabbed by this client. A zero *modifier* mask is equivalent to issuing the request for all possible modifier combinations including the combination of no modifiers. A *button* of **:any** is equivalent to issuing the request for all possible buttons. This has no effect on an active grab.

window — A **window**.

button — The button (type **card8**) that is released or **:any**.

:modifiers — A **modifier-mask**.

Grabbing the Keyboard

12.10 CLX provides the **grab-keyboard** and **ungrab-keyboard** functions for actively grabbing and releasing control of the keyboard.

grab-keyboard *window* &key **:owner-p :sync-pointer-p :sync-keyboard-p :time** Function

Returns:
grab-status — One of **:already-grabbed**, **:frozen**, **:invalid-time**, **:not-viewable**, or **:success**.

Actively grabs control of the keyboard. Further key events are reported only to the grabbing client. The request overrides any active keyboard grab by this client. **grab-keyboard** generates **:focus-in** and **:focus-out** events.

If **:owner-p** is **nil**, all generated key events are reported with respect to *window*. If **:owner-p** is true, then a generated key event that would normally be reported to this client is reported normally; otherwise the event is reported with respect to the *window*. Both **:key-press** and **:key-release** events are always reported, independent of any event selection made by the client.

If **:sync-keyboard-p** is **nil**, keyboard event processing continues normally (asynchronously); if the keyboard is currently frozen by this client, then processing of keyboard events is resumed. If **:sync-keyboard-p** is true, the keyboard (as seen via the protocol) appears to freeze, and no further keyboard events are generated by the server until the grabbing client issues a releasing **allow-events** request. Actual keyboard changes are not lost while the keyboard is frozen; they are simply queued for later processing.

If **:sync-pointer-p** is **nil**, pointer event processing is unaffected by activation of the grab. If **:sync-pointer-p** is true, the pointer (as seen via the protocol) appears to freeze, and no further pointer events are generated by the server until the grabbing client issues a releasing **allow-events** request. Actual pointer changes are not lost while the pointer is frozen; they are simply queued for later processing.

The grab can fail with a status of:

- **:already-grabbed** if the keyboard is actively grabbed by some other client
- **:frozen** if the keyboard is frozen by an active grab from another client
- **:not-viewable** if *window* is not viewable
- **:invalid-time** if the specified time is earlier than the last-keyboard-grab time or later than the current server time. Otherwise, **grab-keyboard** returns a status of **:success** and last-keyboard-grab time is set to the specified time, with current-time replaced by current server time.

window — A **window**.

:owner-p — If true, all client windows receive keyboard input normally.

:sync-pointer-p — Indicates whether the pointer is in synchronous or asynchronous mode.

:sync-keyboard-p — Indicates whether the keyboard is in synchronous or asynchronous mode.

:time — A **timestamp**.

ungrab-keyboard *display* &key **:time** Function

Releases the keyboard if this client has it actively grabbed (from either **grab-keyboard** or **grab-key**), and releases any queued events. The request has no effect if the specified time is earlier than the last-keyboard-grab time or is later than the current server time. An **ungrab-keyboard** is performed automatically if the event window for an active keyboard grab becomes not viewable.

display — A **display**.

:time — A **timestamp**.

Grabbing a Key

12.11 The following paragraphs describe the functions used for passively grabbing and releasing the keyboard.

grab-key *window key* &key (**:modifiers** 0) **:owner-p :sync-pointer-p :sync-keyboard-p :time** Function

This request establishes a passive grab on the keyboard. If the specified *key* (which can also be a modifier key) is pressed (whether or not any specified modifier keys are down), and either of the following is true:

- *window* is an ancestor of (or is) the focus window
- *window* is a descendant of the focus window and contains the pointer
- These constraints are not satisfied for any ancestor of *window*, then the following occurs:
  - The keyboard is actively grabbed as described in **grab-keyboard**
  - The last-keyboard-grab time is set to the time that the *key* was pressed (as transmitted in the **:key-press** event)
  - The **:key-press** event is reported

The interpretation of the remaining arguments is as for **grab-keyboard**. The active grab is terminated automatically when the specified *key* has been released, independent of the state of the modifier keys.

A zero modifier mask is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). It is not required that all specified modifiers have currently assigned keycodes. A *key* of **:any** is equivalent to issuing the request for all possible keycodes. Otherwise, the *key* must be in the range specified by **display-min-keycode** and **display-max-keycode** in the connection setup.

window — A **window**.

key — The key (type **card8**) to be grabbed or **:any**.

:modifiers — A **modifier-mask**.

:owner-p — If true, all client windows receive keyboard input normally.

:sync-pointer-p — Indicates whether the pointer is in synchronous or asynchronous mode.

:sync-keyboard-p — Indicates whether the keyboard is in synchronous or asynchronous mode.

:time — A **timestamp**.

ungrab-key *window key* &key (**:modifiers** 0) Function

Releases the *key* combination on the specified *window* if it was grabbed by this client. A zero modifier mask of **:any** is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). A *key* of **:any** is equivalent to issuing the request for all possible keycodes. **ungrab-key** has no effect on an active grab.

window — A **window**.

key — The key (type **card8**) to be released or **:any**.

:modifiers — A **modifier-mask**.

Event Types

12.12 The following paragraphs contain detailed descriptions of the contents of each event type. In CLX, events are not actually represented by structures, but rather by lists of keyword values passed to handler functions or by values bound to symbols within the clauses of **event-case** and **event-cond** forms. Nevertheless, it is convenient to describe event contents in terms of slots and to identify the components of events with slot name symbols. In fact, CLX uses the **declare-event** macro to define event slot symbols and to map these symbols to specific event data items returned by the X server (see paragraph 12.12.8, Declaring Event Types).

The following paragraphs describe each event type, listing its **event-key** keyword symbol and its slot name symbols. An event keyword symbol identifies a specific event type. An event keyword symbol can be given as an argument to **send-event** or to an event handler function; it can also appear in the *event-match* form of an **event-case** clause. An event slot name symbol identifies a specific event data item. Event slot names appear as keywords with associated values among the arguments passed to **send-event** or to an event handler function; as non-keyword symbols, they can also be in the *event-slots* form of an **event-case** clause.

In certain cases, more than one name symbol is defined for the same event slot. For example, in **:key-press** events, the symbols *window* and *event-window* both refer to the same event data item.

Keyboard and Pointer Events

12.12.1 The keyboard and pointer events are: **:key-press, :key-release, :button-press, :button-release**, **:motion-notify**, **:enter-notify**, and **:leave-notify.**

:key-press, :key-release, :button-press, :button-release Event Type

Selected by: — **:key-press**, **:key-release**, **:button-press**, or **:button-release**.

:key-press, and **:key-release** events are generated when a key or pointer button changes state. Note that **:key-press** and **:key-release** are generated for all keys, even those mapped to modifiers. All of these event types have the same slots. The window containing the pointer at the time of the event is referred to as the *source* window. The *event window* is the window to which the event is actually reported. The event window is found by starting with the source window and looking up the hierarchy for the first window on which any client has selected interest in the event (provided no intervening window prohibits event generation by including the event type in its do-not-propagate-mask). The actual window used for reporting can be modified by active grabs and, in the case of keyboard events, can be modified by the focus window.

A **:button-press** event has the effect of a temporary **grab-button**. When a pointer button is pressed and no active pointer grab is in progress, the ancestors of the source window are searched from the *root* down, looking for a passive grab to activate. If no matching passive grab on the button exists, then an active grab is started automatically for the client receiving the **:button-press** event, and the last-pointer-grab time is set to the current server time. The effect is essentially equivalent to calling **grab-button** with the following arguments:

Argument	Description
window	The event window.
button	The button that was pressed.
event-mask	The client's selected pointer events on the event window.
:modifiers	0
:owner-p	**t** if the client has **:owner-grab-button** selected on the event window; otherwise **nil**.
:sync-pointer-p	**nil**
:sync-keyboard-p	**nil**
:confine-to	**nil**
:cursor	**nil**

The **:button-press** grab is terminated automatically when all buttons are released. The functions **ungrab-pointer** and **change-active-pointer-grab** can both be used to modify the **:button-press** grab.

window, event-window — Type **window**.

The window receiving the event.

code — Type **card8**.

The *code* argument varies with the event type. For **:key-press** and **:key-release**, *code* is the keycode (see paragraph 14.4, Keyboard Encodings). For **:button-press** and **:button-release**, *code* is the pointer button number.

x — Type **int16**.

If *event-window* is on the same screen as root, then *x* and *y* are the pointer coordinates relative to the *event-window*; otherwise *x* and *y* are zero.

y — Type **int16**.

If *event-window* is on the same screen as root, then *x* and *y* are the pointer coordinates relative to the *event-window*; otherwise *x* and *y* are zero.

state — Type **card16**.

A mask that gives the state of the buttons and modifier keys just before the event.

time — Type **card32**.

A timestamp for the moment when the event occurred.

root — Type **window**.

The root window of the source window.

root-x — Type **int16**.

The x coordinate of the pointer position relative to root at the time of the event.

root-y — Type **int16**.

The y coordinate of the pointer position relative to root at the time of the event.

child — Type (**or null window**).

If the source window is an inferior of the *event-window*, *child* is set to the child of *event-window* that is an ancestor of (or is) the source window; otherwise, it is set to **nil**.

same-screen-p — Type **boolean**.

True if *event-window* and root are on the same screen.

:motion-notify Event Type

Selected by: — **:button-1-motion**, **:button-2-motion**, **:button-3-motion**, **:button-4-motion**, **:button-5-motion**, **:button-motion**, or **:pointer-motion**.

The **:motion-notify** event is generated when the pointer moves. A **:motion-notify** event has the same slots as **:button-press, :button-release**, **:key-press**, and **:key-release** events, with the exception that the *code* slot is replaced by the *hint-p* slot. As with these other events, the event window for **:motion-notify** is found by starting with the source window and looking up the hierarchy for the first window on which any client has selected interest in the event (provided no intervening window prohibits event generation by including **:motion-notify** in its do-not-propagate-mask).The actual window used for reporting can be modified by active grabs.

:motion-notify events are generated only when the motion begins and ends in the window. The granularity of motion events is not guaranteed, but a client selecting for motion events is guaranteed to get at least one event when the pointer moves and comes to rest. Selecting **:pointer-motion** generates **:motion-notify** events regardless of the state of the pointer buttons. By selecting some subset of **:button[1-5]-motion** instead, **:motion-notify** events are only received when one or more of the specified buttons are pressed. By selecting **:button-motion**, **:motion-notify** events are only received when at least one button is pressed. If **:pointer-motion-hint** is also selected, the server is free to send only one **:motion-notify**, until either the key or button state changes, the pointer leaves the event window, or the client calls **query-pointer** or **motion-events**.

hint-p — Type **boolean**.

True if the event is a hint generated by selecting **:pointer-motion-hint**.

:enter-notify, :leave-notify Event Type

Selected by: — **:enter-window** or **:leave-window**.

If pointer motion or a window hierarchy change causes the pointer to be in a different window than before, **:enter-notify** and **:leave-notify** events are generated instead of a **:motion-notify** event. All **:enter-notify** and **:leave-notify** events caused by a hierarchy change are generated after any hierarchy event (**:unmap-notify**, **:map-notify**, **:configure-notify**, **:gravity-notify**, or **:circulate-notify**) caused by that change, but the ordering of **:enter-notify** and **:leave-notify** events with respect to **:focus-out**, **:visibility-notify**, and **:exposure** events is not constrained by the X protocol. An **:enter-notify** or **:leave-notify** event can also be generated when a client application calls **change-active-pointer-grab**, **grab-pointer**, or **ungrab-pointer**.

window, event-window — Type **window**.

The window receiving the event.

x — Type **int16**.

The final pointer position. If *event-window* is on the same screen as root, then *x* and *y* are the pointer coordinates relative to the *event-window*; otherwise *x* and *y* are zero.

y — Type **int16**.

The final pointer position. If *event-window* is on the same screen as root, then *x* and *y* are the pointer coordinates relative to the *event-window*; otherwise *x* and *y* are zero.

mode — Type (**member :normal :grab :ungrab**).

Events caused when the pointer is actively grabbed have mode **:grab**. Events caused when an active pointer grab deactivates have mode **:ungrab**. In all other cases, mode is **:normal**.

kind — Type (**member :ancestor :virtual :inferior :nonlinear :nonlinear-virtual**).

When the pointer moves from window A to window B, and A is an inferior of B:

- **:leave-notify** with *kind* **:ancestor** is generated on A
- **:leave-notify** with *kind* **:virtual** is generated on each window between A and B exclusive (in that order)
- **:enter-notify** with *kind* **:inferior** is generated on B

When the pointer moves from window A to window B, and B is an inferior of A:

- **:leave-notify** with *kind* **:inferior** is generated on A
- **:enter-notify** with *kind* **:virtual** is generated on each window between A and B exclusive (in that order)
- **:enter-notify** with *kind* **:ancestor** is generated on B

When the pointer moves from window A to window B, with window C being their least common ancestor:

- **:leave-notify** with *kind* **:nonlinear** is generated on A
- **:leave-notify** with *kind* **:nonlinear-virtual** is generated on each window between A and C exclusive (in that order)
- **:enter-notify** with *kind* **:nonlinear-virtual** is generated on each window between C and B exclusive (in that order)
- **:enter-notify** with *kind* **:nonlinear** is generated on B

When the pointer moves from window A to window B, on different screens:

- **:leave-notify** with *kind* **:nonlinear** is generated on A
- If A is not a root window, **:leave-notify** with *kind* **:nonlinear-virtual** is generated on each window above A up to and including its root (in order)
- If B is not a root window, **:enter-notify** with *kind* **:nonlinear-virtual** is generated on each window from B's root down to but not including B (in order)
- **:enter-notify** with *kind* **:nonlinear** is generated on B

When a pointer grab activates (but after any initial warp into a confine-to window, and before generating any actual **:button-press** event that activates the grab), with *G* the **grab-window** for the grab and *P* the window the pointer is in, then **:enter-notify** and **:leave-notify** events with mode **:grab** are generated (as for **:normal** above) as if the pointer were to suddenly warp from its current position in *P* to some position in *G*. However, the pointer does not warp, and the pointer position is used as both the *initial* and *final* positions for the events.

When a pointer grab deactivates (but after generating any actual **:button-release** event that deactivates the grab), with *G* the **grab-window** for the grab and *P* the window the pointer is in, then **:enter-notify** and **:leave-notify** events with mode **:ungrab** are generated (as for **:normal** above) as if the pointer were to suddenly warp from from some position in *G* to its current position in *P*. However, the pointer does not warp, and the current pointer position is used as both the *initial* and *final* positions for the events.

focus-p — Type **boolean**.

If *event-window* is the focus window or an inferior of the focus window, then *focus-p* is **t**; otherwise, *focus-p* is **nil**.

state — Type **card16**.

A mask that gives the state of the buttons and modifier keys just before the event.

time — Type **card32**.

A timestamp for the moment when the event occurred.

root — Type **window**.

The root window containing the final pointer position.

root-x — Type **int16**.

The x coordinate of the pointer position relative to root at the time of the event.

root-y — Type **int16**.

The y coordinate of the pointer position relative to root at the time of the event.

child — Type (**or null window**).

In a **:leave-notify** event, if a child of the *event-window* contains the initial position of the pointer, the *child* slot is set to that child; otherwise, the *child* slot is **nil**. For an **:enter-notify** event, if a child of the *event-window* contains the final pointer position, the *child* slot is set to that child; otherwise, the *child* slot is **nil**.

same-screen-p — Type **boolean**.

True if *event-window* and root are on the same screen.

Input Focus Events

12.12.2 The input focus events are **:focus-in** and **:focus-out**.

:focus-in, :focus-out — Event Type

Selected by: — **:focus-change**.

:focus-in and **:focus-out** events are generated when the input focus changes. All **:focus-out** events caused by a window **:unmap** are generated after any **:unmap-notify** event, but the ordering of **:focus-out** with respect to generated **:enter-notify**, **:leave-notify**, **:visibility-notify**, and **:expose** events is not constrained.

window, event-window — Type **window**.

For **:focus-in**, the new input focus window. For **:focus-out,** the previous input focus window.

mode — Type (**member :normal :while-grabbed :grab :ungrab**).

Events generated by **set-input-focus** when the keyboard is not grabbed have mode **:normal**. Events generated by **set-input-focus** when the keyboard is grabbed have mode **:while-grabbed**. Events generated when a keyboard grab activates have mode **:grab**, and events generated when a keyboard grab deactivates have mode **:ungrab**.

kind — Type (**member :ancestor :virtual :inferior :nonlinear :nonlinear-virtual :pointer :pointer-root :none**).

When the focus moves from window A to window B, and A is an inferior of B, with the pointer in window P:

- **:focus-out** with *kind* **:ancestor** is generated on A
- **:focus-out** with *kind* **:virtual** is generated on each window between A and B exclusive (in that order)
- **:focus-in** with *kind* **:inferior** is generated on B
- If P is an inferior of B, but P is not A or an inferior of A or an ancestor of A, **:focus-in** with *kind* **:pointer** is generated on each window below B down to and including P (in order)

When the focus moves from window A to window B, and B is an inferior of A, with the pointer in window P:

- If P is an inferior of A, but P is not A or an inferior of B or an ancestor of B, **:focus-out** with *kind* **:pointer** is generated on each window from P up to but not including A (in order)
- **:focus-out** with *kind* **:inferior** is generated on A
- **:focus-in** with *kind* **:virtual** is generated on each window between A and B exclusive (in that order)
- **:focus-in** with *kind* **:ancestor** is generated on B

When the focus moves from window A to window B, with window C being their least common ancestor, and with the pointer in window P:

- If P is an inferior of A, **:focus-out** with *kind* **:pointer** is generated on each window from P up to but not including A (in order)
- **:focus-out** with *kind* **:nonlinear** is generated on A
- **:focus-out** with *kind* **:nonlinear-virtual** is generated on each window between A and C exclusive (in that order)
- **:focus-in** with *kind* **:nonlinear-virtual** is generated on each window between C and B exclusive (in that order)
- :focus-in with *kind* **:nonlinear** is generated on B
- If P is an inferior of B, **:focus-in** with *kind* **:pointer** is generated on each window below B down to and including P (in order)

When the focus moves from window A to window B, on different screens, with the pointer in window P:

- If P is an inferior of A, **:focus-out** with *kind* **:pointer** is generated on each window from P up to but not including A (in order)
- **:focus-out** with *kind* **:nonlinear** is generated on A
- If A is not a root window, **:focus-out** with *kind* **:nonlinear-virtual** is generated on each window above A up to and including its root (in order)
- If B is not a root window, **:focus-in** with *kind* **:nonlinear-virtual** is generated on each window from B's root down to but not including B (in order)
- **:focus-in** with *kind* **:nonlinear** is generated on B
- If P is an inferior of B, **:focus-in** with *kind* **:pointer** is generated on each window below B down to and including P (in order)

When the focus moves from window A to **:pointer-root** (or **:none**), with the pointer in window P:

- If P is an inferior of A, **:focus-out** with *kind* **:pointer** is generated on each window from P up to but not including A (in order)
- **:focus-out** with *kind* **:nonlinear** is generated on A
- If A is not a root window, **:focus-out** with *kind* **:nonlinear-virtual** is generated on each window above A up to and including its root (in order)
- **:focus-in** with *kind* **:pointer-root** (or **:none**) is generated on all root windows
- If the new focus is **:pointer-root**, **:focus-in** with *kind* **:pointer** is generated on each window from P's root down to and including P (in order)

When the focus moves from **:pointer-root** (or **:none**) to window A, with the pointer in window P:

- If the old focus is **:pointer-root**, **:focus-out** with *kind* **:pointer** is generated on each window from P up to and including P's root (in order)
- **:focus-out** with *kind* **:pointer-root** (or **:none**) is generated on all root windows
- If A is not a root window, **:focus-in** with *kind* **:nonlinear-virtual** is generated on each window from A's root down to but not including A (in order)
- **:focus-in** with *kind* **:nonlinear** is generated on A
- If P is an inferior of A, **:focus-in** with *kind* **:pointer** is generated on each window below A down to and including P (in order)

When the focus moves from **:pointer-root** to **:none** (or vice versa), with the pointer in window P:

- If the old focus is **:pointer-root**, **:focus-out** with *kind* **:pointer** is generated on each window from P up to and including P's root (in order)
- **:focus-out** with *kind* **:pointer-root** (or **:none**) is generated on all root windows
- **:focus-in** with *kind* **:none** (or **:pointer-root**) is generated on all root windows
- If the new focus is **:pointer-root**, **:focus-in** with *kind* **:pointer** is generated on each window from P's root down to and including P (in order)

When a keyboard grab activates (but before generating any actual **:key-press** event that activates the grab), with *G* the **grab-window** for the grab and *F* the current focus, then **:focus-in** and **:focus-out** events with mode **:grab** are generated (as for **:normal** above) as if the focus were to change from *F* to *G*.

When a keyboard grab deactivates (but after generating any actual **:key-release** event that deactivates the grab), with *G* the **grab-window** for the grab and *F* the current focus, then **:focus-in** and **:focus-out** events with mode **:ungrab** are generated (as for **:normal** above) as if the focus were to change from *G* to *F*.

Keyboard and Pointer State Events

12.12.3 The keyboard and pointer state events are **:keymap-notify** and **:mapping-notify.**

:keymap-notify Event Type

Selected by: — :**keymap-state**.

The **:keymap-notify** event returns the current state of the keyboard. **:keymap-notify** is generated immediately after every **:enter-notify** and **:focus-in**.

window, event-window — Type **window**.

The window receiving an **:enter-notify** or **:focus-in** event.

keymap — Type (**bit-vector 256**).

A bit-vector containing the logical state of the keyboard. Each bit set to 1 indicates that the corresponding key is currently pressed. The vector is represented as 32 bytes. For *n* from 0 to 7, byte *n* (from 0) contains the bits for keys 8*n* to 8*n*+7, with the least significant bit in the byte representing key 8*n*.

:mapping-notify Event Type

The X server reports **:mapping-notify** events to all clients. There is no mechanism to express disinterest in this event. The X server generates this event type whenever a client application calls one of the following:

- **set-modifier-mapping** to indicate which keycodes to use as modifiers (the status reply must be **:mapping-success**)
- **change-keyboard-mapping** to change the keyboard mapping
- s**et-pointer-mapping** to set the pointer mapping (the status reply must be **:mapping-success**)

request — Type (**member :modifier :keyboard :pointer**).

Indicates the kind of change that occurred—**:modifier** for a successful **set-modifier-mapping**, **:keyboard** for a successful **change-keyboard-mapping**, and **:pointer** for a successful **set-pointer-mapping**.

start — Type **card8**.

If request is **:keyboard**, then *start* and *count* indicate the range of altered keycodes.

count — Type **card8**.

If request is **:keyboard**, then *start* and *count* indicate the range of altered keycodes.

Exposure Events

12.12.4 The X server cannot guarantee that a window's content is preserved when the window is obscured or reconfigured. X requires client applications to be capable of restoring the contents of a previously-invisible window region whenever it is exposed. Therefore, the X server sends events describing the exposed window and its exposed region. For a simple window, a client can choose to redraw the entire content whenever any region is exposed. For a complex window, a client can redraw only the exposed region.

:exposure Event Type

Selected by: — **:exposure**.

An **:exposure** event is sent when redisplay is needed for a window region whose content has been lost. Redisplay is needed when one of the following occurs:

- A region is exposed for a window and the X server has no backing store for the region
- A region of a viewable window is obscured and the X server begins to honor the window's backing-store attribute of **:always** or **:when-mapped**
- The X server begins to honor an unviewable window's backing-store attribute of **:always** or :**when-mapped**.

The regions needing redisplay are decomposed into an arbitrary set of rectangles, and an **:exposure** event is generated for each rectangle. For a given action causing **:exposure** events, the set of events for a given window are guaranteed to be reported contiguously.

:exposure events are never generated for **:input-only** windows.

All **:exposure** events caused by a hierarchy change are generated after any hierarchy event (**:unmap-notify**, **:map-notify**, **:configure-notify**,**:gravity-notify**, or **:circulate-notify**) caused by that change. All **:exposure** events on a given window are generated after any **:visibility-notify** event on that window, but it is not required that all **:exposure** events on all windows be generated after all visibility events on all windows. The ordering of :**exposure** events with respect to **:focus-out**, **:enter-notify**, and **:leave-notify** events is not constrained.

window, event-window — Type **window**.

The window needing redisplay.

x — Type **card16**.

The position of the left edge of the region to redisplay, relative to the *event-window.*

y — Type **card16**.

The position of the top edge of the region to redisplay, relative to the *event-window.*

width — Type **card16**.

The width of the region to redisplay.

height — Type **card16**.

The height of the region to redisplay.

count — Type **card16**.

If count is zero, then no more **:exposure** events for this window follow. If count is nonzero, then at least that many more **:exposure** events for this window follow (and possibly more).

:graphics-exposure Event Type

A **:graphics-exposure** event is generated by a call to **copy-area** or **copy-plane** when the exposures attribute of the graphics context is **:on**. A **:graphics-exposure** event reports a destination region whose content cannot be computed because the content of the corresponding source region has been lost. For example, the missing source region may be obscured or may lie outside the current source drawable size. For a given action causing **:graphics-exposure** events, the set of events for a given destination are guaranteed to be reported contiguously.

drawable, event-window — Type **drawable**.

The destination drawable for the **copy-area** or **copy-plane** function.

x — Type **card16**.

The position of the left edge of the destination region, relative to the *drawable*.

y — Type **card16**.

The position of the top edge of the destination region, relative to the *drawable*.

width — Type **card16**.

The width of the destination region.

height — Type **card16**.

The height of the destination region.

count — Type **card16**.

If count is zero then no more **:graphics-exposure** events for the *drawable* follow. If count is nonzero then at least that many more **:graphics-exposure** events for the *drawable* follow (and possibly more).

major — Type **card8**.

The major opcode for the graphics request generating the event (62 for **copy-area**, 63 for **copy-plane**).

minor — Type **card16**.

The minor opcode for the graphics request generating the event (0 for both **copy-area** and **copy-plane**).

:no-exposure Event Type

A **:no-exposure** event is generated by a call to **copy-area** or **copy-plane** when the exposures attribute of the graphics context is **:on**. If no **:graphics-exposure** events are generated, then a single **:no-exposure** event is sent.

drawable, event-window — Type **drawable**.

The destination drawable for the **copy-area** or **copy-plane** function.

major — Type **card8**.

The major opcode for the graphics request generating the event (62 for **copy-area**, 63 for **copy-plane**).

minor — Type **card16**.

The minor opcode for the graphics request generating the event (0 for both **copy-area** and **copy-plane**).

Window State Events

12.12.5 The following paragraphs describe the events that can be received when a window becomes:

- Created
- Destroyed
- Invisible
- Mapped
- Moved
- Reparented
- Resized
- Restacked
- Unmapped
- Visible

:circulate-notify Event Type

Selected by: — **:structure-notify** on a window or **:substructure-notify** on its parent.

A **:circulate-notify** event is generated whenever a window is actually restacked as a result of a client application calling **circulate-window-up** or **circulate-window-down**.

event-window — Type **window**.

The window receiving the event.

window — Type **window**.

The window that was restacked.

place — Type (**member :top :bottom**).

If place is **:top**, the *window* is now on top of all siblings. Otherwise, it is below all siblings.

:configure-notify Event Type

Selected by: — **:structure-notify** on a window or **:substructure-notify** on its parent.

The **:configure-notify** event is generated when the position or size of a window actually changes as a result of a client application setting its *x*, *y*, *width*, *height*, or *border-width* attributes.

event-window — Type **window**.

The window receiving the event.

window — Type **window**.

The window that was moved or resized.

x — Type **int16**.

x and *y* specify the new upper-left corner position of the *window* relative to its parent.

y — Type **int16**.

x and *y* specify the new upper-left corner position of the *window* relative to its parent.

width — Type **card16**.

width and *height* specify the new size of the *window* interior.

height — Type **card16**.

width and *height* specify the new size of the *window* interior.

border-width — Type **card16**.

The new *window* border width.

above-sibling — Type (**or null window**).

The sibling immediately below the *window.* If above-sibling is **nil**, then the *window* is below all of its siblings.

override-redirect-p — Type **boolean**.

override-redirect-p is true if the override-redirect attribute of the *window* is **:on**; otherwise, it is **nil**. See **window-override-redirect** in paragraph 4.3, Window Attributes.

The X server can report **:create-notify** events to clients wanting information about creation of windows. The X server generates this event whenever a client application creates a window by calling **create-window**.

To receive this event type in a client application, you **setf** the **:substructure-notify** as the event-mask in the parent window's event-mask slot.

:create-notify Event Type

Selected by: **:substructure-notify**.

The **:create-notify** event is generated when a *window* is created and is sent to the *parent* window.

parent, event-window — Type **window**.

The parent window receiving the event.

window — Type **window**.

The new window created.

x — Type **int16**.

x and *y* specify the initial upper-left corner position of the *window* relative to the parent.

y — Type **int16**.

x and *y* specify the initial upper-left corner position of the *window* relative to the parent.

width — Type **card16**.

width and *height* specify the initial size of the *window* interior.

height — Type **card16**.

width and *height* specify the initial size of the *window* interior.

border-width — Type **card16**.

The initial *window* border width.

override-redirect-p — Type **boolean**.

override-redirect-p is true if the override-redirect attribute of the *window* is **:on**; otherwise, it is **nil**. See **window-override-redirect** in paragraph 4.3, Window Attributes.

:destroy-notify Event Type

Selected by — **:structure-notify** on a window or **:substructure-notify** on its parent.

The **:destroy-notify** event is generated when a *window* is destroyed. The ordering of the **:destroy-notify** events is such that for any given window, **:destroy-notify** is generated on all inferiors of a window before **:destroy-notify** is generated on the *window*. The ordering among siblings and across subhierarchies is not otherwise constrained.

event-window — Type **window**.

The window receiving the event.

window — Type **window**.

The window that was destroyed.

:gravity-notify Event Type

Selected by: — **:structure-notify** on a window or **:substructure-notify** on its parent.

The X server can report **:gravity-notify** events to clients wanting information about when a *window* is moved because of a change in the size of its parent. The X server generates this event whenever a client application actually moves a child window as a result of resizing its parent by calling **with-state** with the appropriate arguments set.

event-window — Type **window**.

The window receiving the event.

window — Type **window**.

The window that was moved.

x — Type **int16**.

x and y specify the new upper-left corner position of the *window* relative to its parent.

y — Type **int16**.

x and y specify the new upper-left corner position of the *window* relative to its parent.

:map-notify Event Type

Selected by: — **:structure-notify** on a window or **:substructure-notify** on its parent.

The X server can report **:map-notify** events to clients wanting information about which windows are mapped. The X server generates this event type whenever a client application changes the *window*'s state from unmapped to mapped by calling **map-window** or **map-subwindow**.

To receive this event type, you **setf :structure-notify** as the event-mask on the *window*'s **event-mask** slot. You can also receive this event type by **setf**ing the **:substructure-notify** event-mask on the parent window.

event-window — Type **window**.

The window receiving the event.

window — Type **window**.

The window that was mapped.

override-redirect-p — Type **boolean**.

override-redirect-p is true if the override-redirect attribute of the *window* is **:on**; otherwise, it is **nil**. See **window-override-redirect** in paragraph 4.3, Window Attributes.

:reparent-notify Event Type

Selected by: — **:structure-notify** on a window or **:substructure-notify** on its old or new parent.

The **:reparent-notify** event is generated when a *window* is reparented.

event-window — Type **window**.

The window receiving the event.

window — Type **window**.

The window that was reparented.

parent — Type **window**.

The new parent of the *window*.

x — Type **int16**.

x and y specify the upper-left corner position of the *window* relative to its new *parent*.

y — Type **int16**.

x and y specify the upper-left corner position of the *window* relative to its new *parent*.

override-redirect-p — Type **boolean**.

override-redirect-p is true if the override-redirect attribute of the *window* is **:on**; otherwise, it is **nil**. See **window-override-redirect** in paragraph 4.3, Window Attributes.

:unmap-notify Event Type

Selected by: — **:structure-notify** on a window or **:substructure-notify** on its parent.

The **:unmap-notify** event is generated when a mapped *window* is unmapped.

event-window — Type **window**.

The window receiving the event.

window — Type **window**.

The window that was unmapped.

co*nfigure-p* — Type **boolean**.

configure-p is true if the *window* has a win-gravity attribute of **:unmap**, and the event was generated because *window*'s parent was resized.

:visibility-notify Event Type

Selected by: — **:visibility-change**.

The **:visibility-notify** event is sent when the visibility of a *window* changes. **:visibility-notify** events are never generated on **:input-only** windows. For the purposes of this event, the visibility of the *window* is not affected by its subwindows.

All **:visibility-notify** events caused by a hierarchy change are generated after any hierarchy event caused by that change (for example, **:unmap-notify**, **:map-notify**, **:configure-notify**, **:gravity-notify**, or **:circulate-notify**). Any **:visibility-notify** event on a given window is generated before any **:exposure** events on that window, but it is not required that all **:visibility-notify** events on all windows be generated before all **:exposure** events on all windows. The ordering of **:visibility-notify** events with respect to **:focus-out**, **:enter-notify**, and **:leave-notify** events is not constrained.

window, event-window — Type **window**.

The window that changed in visibility.

state — Type (**member :unobscured :partially-obscured :fully-obscured**).

When the *window* was either unviewable or it was viewable and at least partially obscured, and the *window* changed to viewable and completely unobscured, then *state* is **:unobscured**.

When the *window* was either unviewable or it was viewable and completely obscured, and the *window* changed to viewable and partially obscured, then *state* is **:partially-obscured**.

When the *window* was either unviewable or it was at least partially visible, and the *window* changed to viewable and completely obscured, then *state* is **:fully-obscured**.

Structure Control Events

12.12.6 The following paragraphs describe events used to *redirect* client requests that reconfigure, restack, or map a window. Structure control events are typically used only by window managers and not by ordinary client applications. Structure control events report redirected requests, allowing a window manager to modify the requests before they are actually performed. However, if the override-redirect attribute of a window is **:on**, then no requests are redirected and no structure control events are generated.

:circulate-request Event Type

The **:circulate-request** event is generated when a client application calls **circulate-window-up** or **circulate-window-down** with a window that has the override-redirect attribute **:off**. The *window* argument specifies the window to be restacked, and *place* specifies what the new position in the stacking order should be (either **:top** or **:bottom**).

Selected by: — **:substructure-redirect** on *parent*.

parent, event-window — Type **window**.

The window receiving the event. The receiving client must have selected **:substructure-redirect** on this window.

window — Type **window**.

The window to be restacked.

place — Type **(member :top :bottom)**.

The new stacking priority requested for *window*.

:colormap-notify Event Type

Selected by: — **:colormap-change**.

The **:colormap-notify** event is generated with *new-p* **t** when the *colormap* associated with a *window* is changed, installed, or uninstalled.

window, event-window — Type **window**.

The window receiving the event.

colormap — Type **(or null colormap)**.

The colormap attribute of the window.

new-p — Type **boolean**.

If *new-p* is true, then the *window*'s colormap attribute has changed to the given *colormap*. Otherwise, the *window*'s colormap attribute has not, but the *colormap* has been installed or uninstalled.

installed-p — Type **boolean**.

If *installed-p* is true, then the *colormap* is currently installed.

:configure-request Event Type

Selected by: — **:substructure-redirect** on parent.

The **:configure-request** event is generated when a client program sets the *x*, *y*, *width*, *height*, *border-width* or stacking priority attributes of a window that has the override-redirect attribute **:off**.

parent, event-window — Type **window**.

The window receiving the event. The receiving client must have selected **:substructure-redirect** on this window.

window — Type **window**.

The window to be reconfigured.

x — Type **int16**.

x and *y* specify the requested upper-left corner position of the *window* relative to the parent. If either *x* or *y* is not specified in the value-mask, then it is set to the current window position.

y — Type **int16**.

x and *y* specify the requested upper-left corner position of the *window* relative to the *parent*. If either *x* or *y* is not specified in the *value-mask*, then it is set to the current window position.

width, height — Type **card16**.

width and *height* specify the requested size of the *window* interior. If either *width* or *height* is not specified in the *value-mask*, then it is set to the current window size.

border-width — Type **card16**

The requested *window* border width. If *border-width* is not specified in the *value-mask*, then it is set to the current window *border-width*.

stack-mode — Type **(member :above :below :top-if :bottom-if :opposite)**.

stack-mode and *above-sibling* specify the requested stacking priority of the *window*. If *stack-mode* is not specified in the *value-mask*, then it is set to **:above**.

above-sibling — Type (**or null window**).

stack-mode and *above-sibling* specify the requested stacking priority of the *window*. If *above-sibling* is not specified in the *value-mask*, then it is set to **nil**.

value-mask — Type **mask16**.

Specifies the changed *window* attributes contained in the redirected client request. Each 1 bit specifies that the corresponding attribute was changed.

:map-request Event Type

Selected by: — **:substructure-redirect** on parent.

The **:map-request** event is generated when a client application maps a *window* that has the override-redirect attribute **:off**.

parent, event-window — Type **window**.

The window receiving the event. The receiving client must have selected **:substructure-redirect** on this window.

window — Type **window**.

The window to be mapped.

:resize-request Event Type

Selected by: — **:resize-redirect**.

The **:resize-request** event is generated when a client program sets the *width* or *height* attributes of a *window* that has the override-redirect attribute **:off**.

w*indow, event-window* — Type **window**.

The window to be resized.

width, height — Type **card16**.

width and *height* specify the requested size of the wi*ndow* interior. If either *width* or *height* was unchanged in the client request, then it is set to the current window size.

Client Communications Events

12.12.7 The client communications events discussed in the following paragraphs are: **:client-message**, **:property-notify**, **:selection-clear**, **:selection-request**, and **:selection-notify**.

:client-message Event Type

The **:client-message** event is generated exclusively by client calls to **send-event**. The X server places no interpretation on the *type* or content of *data* sent in a **:client-message**. A client can neither select **:client-message** events nor avoid receiving them.

window, event-window — Type **window**.

The window receiving the event.

type — Type **keyword**.

An xatom keyword that specifies the type of client message. Interpretation of the type is determined solely by agreement between the sending and receiving clients.

format — Type (**member 8 16 32**).

An integer that specifies whether *data* should be viewed as a sequence of 8-bit, 16-bit, or 32-bit quantities.

data — Type **(sequence integer)**.

The data content of the client message. *data* always consists of 160 bytes — depending on format, either 20 8-bit values, 10 16-bit values or 5 32-bit values. The amount of this data actually used by a particular client message depends on the type.

:property-notify Event Type

Selected by: — **:property-change**.

The **:property-notify** event is generated when a window property is changed or deleted.

window, event-window — Type **window**.

The window receiving the event.

atom — Type **keyword**.

The property that was changed or deleted.

state — Type **(member :new-value :deleted)**.

state is **:new-value** if the property was changed using **change-property** or **rotate-properties**, even if zero-length data was added or if all or part of the property was replaced with identical data. *state* is **:deleted** if the property was deleted using **delete-property** or **get-property**.

time — Type **timestamp**.

The server time when the property was changed or deleted.

:selection-clear Event Type

The **:selection-clear** event is reported to the previous owner of a *selection* when the owner of the *selection* is changed. The selection owner is changed by a client using **setf**. A client can neither select **:selection-clear** events nor avoid receiving them.

window, event-window — Type **window**.

The window losing ownership of the *selection*.

selection — Type **keyword**.

The name of the selection.

time — Type **timestamp**.

The last-change time recorded for the *selection*.

:selection-notify Event Type

The **:selection-notify** event is sent to a client calling **convert-selection**. **:selection-notify** reports the result of the client request to return the current value of a *selection* into a particular form. **:selection-notify** is sent using **send-event** by the owner of the selection or (if no owner exists) by the X server. A client can neither select **:selection-notify** events nor avoid receiving them.

NOTE: Standard conventions for inter-client communication require the following additional steps in processing a **:selection-notify** event:

1. The client receiving this event should call **get-property** to return the converted selection value.
2. After receiving the selection value, the property should then be deleted (either by using the **:delete-p** argument to **get-property** or by calling **delete-property**).

---

window, event-window — Type **window**.

The requestor window given in the call to **convert-selection**.

selection — Type **keyword**.

The selection to be converted.

target — Type **keyword**.

An **xatom** specifying the type of the converted selection value. This is the same target type given in the call to **convert-selection**.

property — Type **(or null keyword)**.

The window property containing the converted selection. If the property is **nil**, then either the *selection* has no owner or the owner could not perform the conversion to the *target* type.

time — Type **timestamp**.

The timestamp from the client call to **convert-selection**.

:selection-request Event Type

The **:selection-request** event is reported to the owner of a selection when a client calls **convert-selection**. This event requests the selection owner to convert the current value of a *selection* into a specified form and to return it to the requestor. A client can neither select **:selection-request** events nor avoid receiving them.

The selection owner should respond to a **:selection-request** event by performing the following steps:

1. Convert the current *selection* value to the *target* type.
2. Store the converted selection value in the *property*. If *property* is **nil**, then the owner should choose the *property*.
3. Call **send-event** to send a **:selection-notify** event to the *requestor* containing the *property* with the converted value. If the *selection* could not be converted to the *target* type, then a **nil** *property* should be sent. The **:selection**, **:target**, and **:time** arguments to **send-event** should be the same as those received in the **:selection-request** event. The event-mask argument to **send-event** should be **nil**; that is, the **:selection-notify** event should be sent to client that created the *requestor*.

NOTE: Standard conventions for inter-client communication require the following additional steps in processing a **:selection-request** event:

1. The property used to store the selection value must belong to the requestor.
2. If the property is **nil**, the target type **atom** should be used as the property name.
3. If the window did not actually own the selection at the given time, the request should be refused, just as if it could not be converted to the target type.

---

window, event-window — Type **window**.

The selection owner receiving the event.

requestor — Type **window**.

The window requesting the converted *selection*.

selection — Type **keyword**.

The selection to be converted.

target — Type **keyword**.

An **xatom** specifying the type of the converted *selection* value.

property — Type **(or null keyword)**.

A requestor window property.

time — Type **timestamp**.

The timestamp sent in the client **convert-selection** request.

Declaring Event Types

12.12.8 CLX uses the **declare-event** macro to define the event slot symbols that access the contents of X events. Most client applications do not need to use **declare-event** because the declarations for all core X events are already defined by CLX. Programmers using extensions to the X protocol can use **declare-event** to allow CLX to handle new event types returned by an extended X server.

declare-event *event-codes* &rest *slot-declarations* Macro

Defines a mapping between event slot symbols and the data items in event messages received from an X server.

The *event-codes* argument gives the event type keyword for the event described. If several event types share the same slots, then *event-codes* can be a list of event type keywords. *slot-declarations* is a list containing an element for each event data item. The order of *slot-declarations* corresponds to the order of event data items defined by the X protocol.

Each element of *slot-declarations* is a list of the form (*type slot-name**), where *type* is a Common Lisp type specifier and *slot-name* is a slot name symbol. The effect of such a list is to declare that the next data items in the event have the given data *type* and are associated with the given *slot-name* symbols. *slot-name* can also be a list of slot name symbols; in this case, each symbol in the list is an alias that refers to the same event data item.

event-codes — An event type keyword or a list of event type keywords.

slot-declarations — A list of clauses defining event slot symbols.

Releasing Queued Events

12.13 A client grabbing the keyboard or pointer can freeze the reporting of events on that device. When an input device is thus frozen, the server queues events until explicitly requested to release them by the grabbing client. CLX programs can use the **allow-events** function to release queued events from a frozen input device.

allow-events *display mode* &optional *time* Function

Releases some queued events if the client has caused a device to freeze. The request has no effect if the *time* is earlier than the last-grab time of the most recent active grab for the client, or if the *time* is later than the current server time. If *time* is **nil**, the current server time is used. The effect of this function depends on the specified *mode*.

- **:async-pointer** — If the pointer is frozen by the client, pointer event processing continues normally. If the pointer is frozen twice by the client on behalf of two separate grabs, **:async-pointer** releases events for both grab*s*. **:async-pointer** has no effect if the pointer is not frozen by the client, but the pointer need not be grabbed by the client.

- **:sync-pointer** — If the pointer is frozen and actively grabbed by the client, pointer event processing continues normally until the next **:button-press** or **:button-release** event is reported to the client, at which time the pointer again appears to freeze. However, if the reported event causes the pointer grab to be released, the pointer does not freeze. **:sync-pointer** has no effect if the pointer is not frozen by the client, or if the pointer is not grabbed by the client.

- **:replay-pointer** — If the pointer is actively grabbed by the client and is frozen as the result of an event having been sent to the client (either from the activation of a **grab-button**, or from a previous **allow-events** with mode **:sync-pointer**, but not from a **grab-pointer**), the pointer grab is released and that event is completely reprocessed, but this time ignoring any passive grabs at or above (towards the root) the **grab-window** of the grab just released. The request has no effect if the pointer is not grabbed by the client, or if the pointer is not frozen as the result of an event.

- **:async-keyboard** — If the keyboard is frozen by the client, keyboard event processing continues normally. If the keyboard is frozen twice by the client on behalf of two separate grabs, **:async-keyboard** releases events for both grabs. **:async-keyboard** has no effect if the keyboard is not frozen by the client, but the keyboard need not be grabbed by the client.

- **:sync-keyboard** — If the keyboard is frozen and actively grabbed by the client, keyboard event processing continues normally until the next **:key-press** or **:key-release** event is reported to the client, at which time the keyboard again appears to freeze. However if the reported event causes the keyboard grab to be released, the keyboard does not freeze. **:sync-keyboard** has no effect if the keyboard is not frozen by the client, or if the keyboard is not grabbed by the client.

- **:replay-keyboard** — If the keyboard is actively grabbed by the client and is frozen as the result of an event having been sent to the client (either from the activation of a grab-key, or from a previous **allow-events** with mode **:sync-keyboard**, but not from a **grab-keyboard**), the keyboard grab is released and that event is completely reprocessed, but this time ignoring any passive grabs at or above (towards the root) the **grab-window** of the grab just released. The request has no effect if the keyboard is not grabbed by the client, or if the keyboard is not frozen as the result of an event.

- **:sync-both** — If both pointer and keyboard are frozen by the client, event processing (for both devices) continues normally until the next **:button-press**, **:button-release**, **:key-press**, or **:key-release** event is reported to the client for a grabbed device (button event for the pointer, key event for the keyboard). At this time, the devices again appear to freeze. If the reported event causes the grab to be released, the devices do not freeze. However, if the other device is still grabbed, then a subsequent event for it will still cause both devices to freeze. **:sync-both** has no effect unless both pointer and keyboard are frozen by the client. If the pointer of keyboard is frozen twice by the client on behalf of two separate grabs, **:sync-both** *thaws* for both, but a subsequent freeze for **:sync-both** will only freeze each device once.

- **:async-both** — If the pointer and the keyboard are frozen by the client, event processing for both devices continues normally. If a device is frozen twice by the client on behalf of two separate grabs, **:async-both** *thaws* for both. **:async-both** has no effect unless both pointer and keyboard are frozen by the client.

:async-pointer, **:sync-pointer**, and **:replay-pointer** have no effect on processing of keyboard events. **:async-keyboard**, **:sync-keyboard**, and **:replay-keyboard** have no effect on processing of pointer events.

It is possible for both a pointer grab and a keyboard grab to be active simultaneously by the same or different clients. When a device is frozen on behalf of either grab, no event processing is performed for the device. It is possible for a single device to be frozen due to both grabs. In this case, the freeze must be released on behalf of both grabs before events can again be processed.

display — A **display**.

mode — One of: **:async-pointer**, **:sync-pointer**, **:reply-pointer**, **:async-keyboard**, **:sync-keyboard**, **:replay-keyboard**, **:async-both**, **:sync-both**.

time — A **timestamp**.

RESOURCES 13

Introduction

13.1 Users need a way to specify preferences for various user interface values (for example, colors, fonts, title strings, and so forth). Applications need a consistent method for determining the default interface values that are specific to them. It is also useful if application interface values can be modified by users without changes to the program code. For example, this capability can make it easy to change the color scheme of a user interface. In CLX, such interface values are referred to as *resources*. CLX defines functions for storing and retrieving interface resources from a resource database. A user can store various user interface values as resources in a resource database; a CLX application can then read these resource values and modify its user interface accordingly.

NOTE: The general term *resource* refers to any application user interface value stored in a resource database. The term *server resource* is used more specifically to refer to the types of objects allocated by an X server and referenced by clients (for example, windows, fonts, graphics contexts, and so forth).

Resource Bindings

13.2 Conceptually, a resource database is a set of resource name-value pairs (or *resource bindings*). The name in a resource binding is a list that is the concatenation of a *path list* and an *attribute name*.

A path list is a list of symbols (or strings) that corresponds to a path through a tree-structured hierarchy. For example, the path:

```
'(top middle bottom)
```

corresponds to a three-level hierarchy in which `middle` is the child of `top`, and `bottom` is the child of `middle`.

Typically, the path of a resource name corresponds to a path in a hierarchy of windows, and each symbol/string names a window in the hierarchy. However, the first element of the path can also represent the overall name of the entire program, and subsequent path elements can refer to an application-specific hierarchy of resource names not strictly related to windows. In addition, a resource name can contain a partially-specified path list. The asterisk symbol (*) is a wildcard that can correspond to any sequence of levels in the hierarchy (including the null sequence). For example, the path:

```
'(top * bottom)
```

corresponds to a hierarchy of two or more levels in which `top` is at the top level and `bottom` is at the bottom level. An element of a path list can be the name of an individual window or the name of a class of windows.

The final element of a resource name list is an attribute name. This symbol (or string) identifies a specific attribute of the object(s) named by the preceding path list. The attribute name can also be the symbol * or the string "*", in which case the resource name refers to all attributes of the path object(s). However, this form of resource name is rarely useful.

Some examples of resource bindings are shown below. In these examples, assume that `mail` is the resource name of a mail reading application. `mail` uses a window of the class `button` whose name is `reply`.

Resource Name	Resource Value
`(mail screen-1 reply background)`	`'green`
`(mail * background)`	`'red`
`(* button background)`	`'blue`

These resource bindings specify the following:

- The `background` attribute resource of `mail` application's `reply` button has the value of `green` on `screen-1`.
- The `background` attribute for the rest of the `mail` application is always `red` on all screens.
- In general, the `background` attribute for all `button` windows is `blue`.

Basic Resource Database Functions

13.3 A **resource-database** structure is a CLX object that represents a set of resource bindings. The following paragraphs describe the CLX functions used to:

- Create a resource database
- Add a resource binding
- Remove a resource binding
- Merge two resource databases
- Map a function over the contents of a resource database

make-resource-database Function

Returns:
resource-database — Type **resource-database**.

Returns an empty resource database.

add-resource *database name-list value* Function

Adds the resource binding specified by *name-list* and *value* to the given *database*. Only one value can be associated with the *name-list* in the *database*. This function replaces any value previously associated with the *name-list*.

database — The **resource-database** for the new resource binding.

name-list — A list containing strings or symbols specifying the name for the resource binding.

value — The value associated with the *name-list* in the resource binding. This can be an object of any type.

delete-resource *database name-list* Function

Removes the resource binding specified by *name-list* from the given *database*.

database — The **resource-database** containing the resource binding.

name-list — A list containing strings or symbols specifying the name for the deleted resource binding.

map-resource *database function* &rest *args* Function

Calls the function for each resource binding in the *database*. For each resource binding consisting of a *name-list* and a *value*, the form (**apply** *function name-list value args*) is executed.

database — A **resource-database**.

function — A **function** object or function symbol.

args — A list of arguments to the *function*.

merge-resources *from-database to-database* Function

Returns:
to-database —Type **resource-database**.

Merges the contents of the *from-database* with the *to-database*. **map-resource** invokes **add-resource** in order to add each resource binding in the *from-database* to the *to-database*. The updated *to-database* is returned.

from-database — The **resource-database** from which resource bindings are read.

to-database — The **resource-database** to which resource bindings are added.

Accessing Resource Values

13.4 The power and flexibility of resource management is the result of the way resource values in a resource database are accessed. A resource binding binding stored in the database generally contains only a partial resource name consisting of a mixture of name and class identifiers and wildcard elements (that is, *). To look up a resource value, an application program starts with two resource name lists of the same length containing no wildcard elements — a *complete resource name* and a *complete resource class*. The lookup algorithm returns the value for the resource binding whose resource name is the closest match to the complete name and class given. The definition of *closest match* takes into account the top-down, parent-child hierarchy of resource names and also the distinction between individual names and class names.

Complete Names and Classes

13.4.1 A resource binding contains a resource name list that can contain names, class names, or a mixture of both. A class name is a symbol or string that represents a group of related objects. The set of names used as class names are not specified by CLX. Instead, class names are defined by agreement between those who use class names when creating resource bindings (that is, users) and those who use class names when accessing resource values (that is, application programmers).

In order to access a value in a resource database, an application uses a key consisting of two items: a *complete resource name* and a *complete resource class*. A complete resource name is a resource name list containing no wildcard elements. A complete resource class is a list of exactly the same form. The distinction between a complete resource name and a complete resource class lies in how they are used to access resource bindings. The elements of a complete resource name are interpreted as names of individual objects; the elements of a complete resource class are interpreted as names of object classes. The complete resource name and class lists used in a resource database access must have the same length.

Like any resource name list, a complete resource name consists of a path list and an attribute name. The first path list element is typically a symbol (or string) identifying the application as a whole. The second element can be a screen root identifier. Subsequent elements can be identifiers for each ancestor window of an application window. Thus, a path list typically identifies a specific window by tracing a path to it through the application window hierarchy. The final element of a complete resource name (its attribute name) is typically the name of a specific attribute of the window given by the path list (for example, `'background`). An attribute name can refer to a feature associated with the window by the application but not by the X server (for example, a font identifier). Similarly, a complete resource class typically represents a path to a window in the application window hierarchy and a specific window attribute. However, a complete resource class contains the class name for each window and for the window attribute.

For instance, in the previous example, the `mail` application can attempt to look up the value of the `background` resource for the `reply button` window by using the following complete resource name:

```
(mail screen-1 reply background)
```

and the following complete resource class:

```
(application root button fill)
```

This complete resource name contains a path list identifying the reply button window — `(mail screen-1 reply)` — and an attribute name for the window background. The corresponding resource class contains the class names for the same path list and window attribute.

Matching Resource Names

13.4.2 The resource lookup algorithm searches a specified resource data base and returns the value for the resource binding whose resource name is the closest match to a given complete resource name and class. The intent of the lookup algorithm is to formalize an intuitive notion of the closest match.

Precedence is given to a match which begins *higher* in the parent-child contact hierarchy. This allows a resource binding with a partial name to define a resource value shared by all members of a window subtree. For example, suppose the resource database contained the following resource bindings:

Resource Name	Resource Value
`(mail * background)`	`'red`
`(* reply background)`	`'blue`

Suppose an application program searched by using the following complete resource name:

```
(mail screen-1 reply background)
```

then the closest matching value returned would be `'red`.

Precedence is given to the more specific match. A name match is more specific than a class match. Either a name or class match is more specific than a wildcard match. For example, suppose the resource database contained the following resource bindings:

Resource Name	Resource Value
`(mail * background)`	`'red`

```
(mail * fill)                          'blue
```

Suppose an application program searched by using the following complete resource name and complete resource class:

```
(mail screen-1 reply background)
(application root button fill)
```

then the closest matching value returned would be `'red`. However, suppose the resource database contained the following resource bindings:

Resource Name	Resource Value
`(mail * background)`	`'red`
`(mail * button background)`	`'blue`

then the closest matching value returned would be `'blue`.

Resource Access Functions

13.4.3 The following paragraphs describe the CLX functions used to return a value from a resource database.

get-resource *database attribute-name attribute-class path-name path-class* Function

Returns:
value — Type **t**.

Returns the value of the resource binding in the *database* whose resource name most closely matches the complete resource name/class given by the *path-name*, *path-class*, *attribute-name*, and *attribute-class*. The lookup algorithm implements the precedence rules described previously to determine the closest match. When comparing name elements, case is significant only if both elements are strings; otherwise, element matching is case-insensitive.

database — A **resource-database**.

attribute-name — A string or symbol giving an attribute name from a complete resource name.

attribute-class — A string or symbol giving an attribute class name from a complete resource class.

path-name — The path list from a complete resource name. *path-name* and *path-class* must have the same length.

path-class — The path list from a complete resource class. *path-name* and *path-class* must have the same length.

get-search-table *database path-name path-class* Function

Returns:
search-table — Type **list**.

Returns a table containing the subset of the *database* that matches the *path-name* and *path-class*. Resources using the same *path-name* and *path-class* can be accessed much more efficiently by using this table as an argument to **get-search-resource**.

database — A **resource-database**.

path-name — The path list from a complete resource name. *path-name* and *path-class* must have the same length.

path-class — The path list from a complete resource class. *path-name* and *path-class* must have the same length.

get-search-resource *table attribute-name attribute-class* Function

Returns:
value — Type **t**.

Returns the value of the resource binding in the search *table* that most closely matches the *attribute-name* and *attribute-class*. The *table* is computed by **get-search-table** and represents a set of resource bindings. The closest match is determined by the same algorithm used in **get-resource**.

The following two forms are functionally equivalent:

```
(get-resource
  database attribute-name attribute-class path-name path-class)

(get-search-resource
 (get-search-table database path-name path-class)
 attribute-name attribute-class)
```

However, the hard part of the search is done by **get-search-table**. Looking up values for several resource attributes that share the same path list can be done much more efficiently with calls to **get-search-resource**.

table — A search table returned by **get-search-table**.

attribute-name — A string or symbol giving an attribute name from a complete resource name.

attribute-class — A string or symbol giving an attribute class name from a complete resource class.

Resource Database Files

13.5 X users and application programs can save resource bindings in a file, using a standard file format shared by all X clients. The following paragraphs describe the CLX functions used to convert between the standard external format of resource files and the internal resource-database format used by application programs.

read-resources *database pathname* &key **:key :test :test-not** Function

Returns:
database — Type **resource-database**.

Reads resource bindings from a resource file in standard X11 format and merges them with the given resource *database*. The **:key** function is called to convert a file resource value into the value stored in the *database*. By default, **:key** is **#'identity**. The **:test** and **:test-not** functions are predicates that select resource bindings to merge, based on the result of the **:key** function. For each file resource binding consisting of a *resource-name* and a *resource-value*, the **:test** (or **:test-not**) function is called with the arguments *resource-name* and (**funcall** *key resource-value*).

database — The **resource-database** to merge.

pathname — A pathname for the resource file to read.

:key — A function used to convert a value from the resource file into a resource binding value.

:test, **:test-not** — Functions used to select which resource bindings from the resource file are merged with the *database*.

write-resources *database pathname* &key **:write :test :test-not** Function

Writes resource bindings found in the *database* to the file given by the *pathname*. The output file is written in the standard X11 format. The **:write** function is used for writing resource values; the default is **#'princ**. The **:write** function is passed two arguments: a *resource-value* and a *stream*. The **:test** and **:test-not** functions are predicates which select resource bindings to write. For each resource binding consisting of a *resource-name* and a *resource-value*, the **:test** (or **:test-not**) function is called with the arguments *resource-name* and *resource-value*.

database — The **resource-database** to write.

pathname — A pathname of the file to write.

:write — A function for writing resource values.

:test, **:test-not** — Functions used to select which resource bindings from the resource file are merged with the *database*.

CONTROL FUNCTIONS 14

Grabbing the Server

14.1 Certain cases may require that a client demand exclusive access to the server, causing the processing for all other clients to be suspended. Such exclusive access is referred to as *grabbing the server*. CLX provides functions to grab and release exclusive access to the server. These function should be used rarely and always with extreme caution, since they have the potential to disrupt the entire window system for all clients.

grab-server *display* Function

Disables processing of requests and close-downs on all connections other than the one on which this request arrived.

display — A **display**.

ungrab-server *display* Function

Restarts processing of requests and close-downs on other connections.

display — A **display**.

with-server-grabbed *display* &body *body* Macro

Grabs the *display* server only within the dynamic extent of the *body*. **ungrab-server** is automatically called upon exit from the *body*. This macro provides the most reliable way for CLX clients to grab the server.

display — A **display**.

body — The forms to execute while the server is grabbed.

Pointer Control

14.2 The following paragraphs describe the CLX functions used to:

- Return or change the pointer acceleration and acceleration threshold
- Return or change the mapping of pointer button numbers

change-pointer-control *display* &key **:acceleration :threshold** Function

Changes the acceleration and/or the acceleration threshold of the pointer for the *display*. The **:acceleration** number is used as a multiplier, typically specified as a rational number of the form *C/P*, where *C* is the number of pixel positions of cursor motion displayed for *P* units of pointer device motion. The acceleration only occurs if the pointer moves more that **:threshold** pixels at once, and only applies to the motion beyond the **:threshold**. Either **:acceleration** or **:threshold** can be set to **:default**, that restores the default settings of the server.

display — A **display**.

:acceleration — A number for the acceleration ratio.

:threshold — The number of pixels required for acceleration to take effect.

pointer-control *display* Function

Returns:
acceleration, *threshold* — Type **number**.

Returns the acceleration and threshold for the *display* pointer.

display — A **display**.

pointer-mapping *display* &key (**:result-type 'list**) Function

Returns:
mapping — Type **sequence** or **card8**.

Returns or (with **setf**) changes the mapping of button numbers for the *display* pointer. The **:result-type** is not used when changing the mapping. If element *i* of the mapping sequence is *j*, then the events from pointer button *j* are reported by the server as events for button *i*+1. (Note that pointer buttons are numbered beginning with one, while the mapping sequence itself is indexed normally from zero.) If element *i* of the mapping sequence is zero, then button *i*+1 is disabled and can no longer generate input events. No two elements of the mapping can have the same non-zero value.

The length of the mapping sequence indicates the actual number of buttons on the device. When changing the mapping, the new mapping must have this same length.

display — A **display**.

:result-type — The type of sequence to return.

Keyboard Control

14.3 The following paragraphs describe the CLX functions used to:

- Return or change keyboard controls
- Ring the keyboard bell
- Return or change the mapping of modifiers
- Return the current up/down state of all keys

bell *display* &optional (*percent-from-normal* 0) Function

Rings the bell on the keyboard at a volume relative to the base volume for the keyboard, if possible. Percent can range from –100 to 100 inclusive, or else a Value error occurs. The following is the bell volume when percent is non-negative:

(– (+ *base percent*) (**quotient** (* *base percent*) 100))

and when percent is negative:

(+ *base* (**quotient** (* *base percent*) 100))

display — A **display**.

percent-from-normal — An integer (–100 through 100).

change-keyboard-control *display* &key **:key-click-percent :bell-percent** Function
:bell-pitch :bell-duration :led :led-mode :key :auto-repeat-mode

Changes the various aspects of the keyboard. The keyword arguments specify which controls to change.

The **:key-click-percent** keyword sets the volume for key clicks, if possible. A value of 0 implies off, while a value of 100 implies loud. Setting **:key-click-percent** to **:default** restores the default value.

The **:bell-percent** sets the base volume for the bell between 0 (off) and 100 (loud) if possible. Setting **:bell-percent** to **:default** restores the default value.

The **:bell-pitch** sets the pitch (specified in Hz) of the bell, if possible. Setting the **:bell-pitch** to **:default** restores the default value. The **:bell-duration** sets the duration (specified in milliseconds) of the bell, if possible. Setting **:bell-pitch** to **:default** restores the default. Note that a bell generator connected with the console but not directly on the keyboard is treated as if it were part of the keyboard.

If both **:led-mode** and **:led** are specified, then the state of that LED is changed, if possible. If only **:led-mode** is specified, the state of all LEDs are changed, if possible. At most 32 LEDs are supported, numbered from one. No standard interpretation of the LEDs are defined.

If both **:auto-repeat-mode** and **:key** are specified, the auto-repeat mode of that key is changed, if possible. If only **:auto-repeat-mode** is specified, the global auto-repeat mode for the entire keyboard is changed, if possible, without affecting the per-key settings. An error occurs if **:key** is specified without **:auto-repeat-mode**.

display — A **display**.

:key-click-percent — An integer (0 100).

:bell-percent — An integer (0 100).

:bell-pitch — A **card16**.

:bell-duration — A **card16**.

:led — A **card8**.

:led-mode — Either **:on** or **:off**.

:key — A **card8** keycode.

:auto-repeat-mode — Either **:on**, **:off**, or **:default**.

keyboard-control *display* Function

Returns:
key-click-percent, bell-percent — Type **card8**.
bell-pitch bell-duration — Type **card16**.
led-mask — Type **card32**.
global-auto-repeat — Either **:on** or **:off**.
auto-repeats — Type **bit-vector**.

Returns the current control values for the keyboard. For the LEDs, the least significant bit of *led-mask* corresponds to LED one, and each one bit in *led-mask* indicates an LED that is lit. *auto-repeats* is a bit vector; each one bit indicates that auto-repeat is enabled for the corresponding key. The vector is represented as 32 bytes. Byte *n* (from 0) contains the bits for keys 8*n* to 8*n*+7, with the least significant bit in the byte representing key 8*n*.

display — A **display**.

modifier-mapping *display* Function

Returns:
shift-keycodes, *lock-keycodes*, *control-keycodes*, *mod1-keycodes*, *mod2-keycodes*, *mod3-keycodes*, *mod4-keycodes*, *mod5-keycodes* — Type **list** of **card8**.

Returns the set of keycodes used for each modifier on the *display* keyboard. Each return value is a list of the **card8** keycodes used for each modifier key. The order of keycodes within each list is server-dependent.

display — A **display**.

query-keymap *display* Function

Returns:
keymap — Type **bit-vector** 256.

Returns a bit vector that describes the state of the keyboard. Each one bit indicates that the corresponding key is currently pressed. The vector is represented as 32 bytes. Byte *n* (from 0) contains the bits for keys 8*n* to 8*n*+7, with the least significant bit in the byte representing key 8*n*.

display — A **display**.

set-modifier-mapping *display* &key **:shift :lock :control :mod1 :mod2 :mod3 :mod4 :mod5** Function

Returns:
status — One of **:success**, **:failed**, or **:device-busy**.

Changes the set of keycodes mapped to the specified modifier keys on the *display* keyboard. Each keyword argument contains a sequence of new **card8** keycodes for a specific modifier. The return value indicates whether the change was completed successfully.

A status of **:failed** is returned if hardware limitations prevent the requested change. For example, multiple keycodes per modifier may not be supported, up transitions on a given keycode may not be supported, or autorepeat may be mandatory for a given keycode. If **:failed** is returned, the mappings for all modifiers remain unchanged.

A status of **:device-busy** is returned if a new keycode given for a modifier was not previously mapped to that modifier and is currently in the down state. In this case, the mappings for all modifiers remain unchanged.

display — A **display**.

:shift, **:lock**, **:control**, **:mod1**, **:mod2**, **:mod3**, **:mod4**, **:mod5** — A sequence of **card8** keycodes for the given modifier.

Keyboard Encodings

14.4 Handling the great diversity of keyboard devices and international language character encodings is a difficult problem for interactive programs that need to receive text input but must also be portable. The X Window System solves this problem by using different sets of encodings for device keys (*keycodes*) and for character symbols (*keysyms*). Each X server maintains a *keyboard mapping* that associates keycodes and keysyms, and which can be returned or changed by client programs.

To handle text input, a CLX client program must follow these steps:

1. Receive a **:key-press** (or **:key-release**) event containing a keycode.

2. Convert the keycode into its corresponding keysym, based on the current keyboard mapping. See **keycode-keysym**.

3. Convert the keysym into the corresponding Common Lisp character. See **keysym-character**.

Keycodes and Keysyms

14.4.1 A *keycode* represents a physical (or logical) key. In CLX, keycodes are values of type (**integer** 8 255). A keycode value carries no intrinsic information, although server implementors may attempt to encode geometry (for example, matrix) information in some fashion so it can be interpreted in a server- dependent fashion. The mapping between keys and keycodes cannot be changed.

A *keysym* is an encoding of a symbol on the cap of a key. In CLX, keysyms are values of type **card32**. The set of defined keysyms include the ISO Latin character sets (1-4), Katakana, Arabic, Cyrillic, Greek, Technical, Special, Publishing, APL, Hebrew, and miscellaneous keys found on keyboards (RETURN, HELP, TAB, and so on). The encoding of keysyms is defined by the X Protocol.

A list of keysyms is associated with each keycode. The length of the list can vary with each keycode. The list is intended to convey the set of symbols on the corresponding key. By convention, if the list contains a single keysym and if that keysym is alphabetic and case distinction is relevant, then it should be treated as equivalent to a two-element list of the lowercase and uppercase keysyms. For example, if the list contains the single keysym for uppercase A, the client should treat it as if it were a pair with lowercase as the first keysym and uppercase A as the second keysym.

For any keycode, the first keysym in the list should be chosen as the interpretation of a key press when no modifier keys are down. The second keysym in the list normally should be chosen when the **:shift** modifier is on, or when the **:lock** modifier is on and **:lock** is interpreted as **:shift-lock**. When the **:lock** modifier is on and is interpreted as **:caps-lock**, it is suggested that the **:shift** modifier first be applied to choose a keysym, but if that keysym is lowercase alphabetic, the corresponding uppercase keysym should be used instead.

Other interpretations of **:caps-lock** are possible; for example, it may be viewed as equivalent to **:shift-lock**, but only applying when the first keysym is lowercase alphabetic and the second keysym is the corresponding uppercase alphabetic. No interpretation of keysyms beyond the first two in a list is suggested here. No spatial geometry of the symbols on the key is defined by their order in the keysym list, although a geometry might be defined on a vendor-specific basis. The X server does not use the mapping between keycodes and keysyms. Rather, the X server stores the mapping merely for reading and writing by clients.

Keyboard Mapping

14.4.2 The X server maintains a keyboard mapping that associates each keycode with one or more keysyms. The following paragraphs describe the CLX functions used to return or change the mapping of keycodes.

change-keyboard-mapping *display keysyms* &key (**:start** 0) **:end** (**:first-keycode :start**) — Function

Changes the mapping of keycodes to *keysyms*. A **:mapping-notify** event is generated for all clients.

The new *keysyms* are specified as a two-dimensional array in which:

(**aref** *keysyms* (+ **:start** *i*) *j*)

is *keysym j* associated with keycode (+ **:first-keycode** *i*). The maximum number of *keysyms* associated with any one keycode is given by:

(**array-dimension** *keysyms* 1)

keysyms should contain **nil** elements to represent those keysyms that are undefined for a given keycode. **:start** and **:end** define the subsequence of the *keysyms* array that defines the new mapping, and the number of keycode mappings changed. By default, **:end** is given by:

(**array-dimension** *keysyms* 0)

The keycodes whose mappings are changed are given by **:first-keycode** through the following:

(+ **:first-keycode** (– **:end :start**) –1)

keycodes outside this range of are not affected. **:first-keycode** must not be less than (**display-min-keycode** *display*), and the last keycode modified must not be greater than (**display-max-keycode** *display*).

display — A **display**.

keysyms — A two-dimensional array of keysym (**card32**) values.

:start, **:end** — Indexes for the subsequence of *keysyms* used.

:first-keycode — A **card8** defining the first keycode mapping changed.

keyboard-mapping *display* &key **:first-keycode :start :end :data** — Function

Returns:
mappings — Type (**array card32** (* *)).

Returns the keysyms mapped to the given range of keycodes for the *display* keyboard. The mappings are returned in the form of a two-dimensional array of **card32** keysym values. The **:data** argument, if given, must be a two-dimensional array in which the returned mappings will be stored. In this case:

(**array-dimension :data** 1)

defines the maximum number of keysyms returned for any keycode. Otherwise, a new array is created and returned.

Upon return:

(**aref** *mappings* (+ *:start i*) *j*)

will contain keysym *j* associated with keycode (+ **:first-keycode i**) (or **nil**, if keysym *j* is undefined for that keycode).

:first-keycode specifies the first keycode whose mapping is returned; by default, **:first-keycode** is (**display-min-keycode** *display*). **:start** and **:end** define the subsequence of the returned array in which the returned mappings are stored. By default, **:start** is given by **:first-keycode** and **:end** is given by:

(1+ (**display-max-keycode** *display*))

:first-keycode must not be less than (**display-min-keycode** *display*), and the last keycode returned must not be greater than (**display-max-keycode** *display*).

display — A **display**.

:first-keycode — A **card8** defining the first keycode mapping returned.

:start, **:end** — Indexes for the subsequence of the returned array which is modified.

:data — If given, a two-dimensional array to receive the returned keysyms.

Using Keycodes and Keysyms

14.4.3 The following paragraphs describe the CLX functions used to:

- Convert a keycode into a keysym
- Convert a keysym into a character

keycode-keysym *display keycode keysym-index* — Function

Returns:
keysym — Type **keysym**.

Returns the *keysym* at the given *keysym-index* from the keysym list for the *keycode* in the current keyboard mapping for the *display* server.

display — A **display**.

keycode — A **card8**.

keysym-index — A **card8**.

keycode-character *display keysym* &optional (*state* 0) — Function

Returns:
character — Type **character** or **null**.

Returns the *character* associated with the *keysym* and the *state*. The *state* is a **mask16** bit mask representing the state of the *display* modifier keys and pointer buttons. See **state-mask-key** in paragraph 1.6, Data Types. If the *keysym* does not represent a Common Lisp character, then **nil** is returned.

The *state* determines the bits attribute of the returned *character*, as follows:

:control	**char-control-bit**
:mod-1	**char-meta-bit**
:mod-2	**char-super-bit**
:mod-3	**char-hyper-bit**

display — A **display**.

keysym — A **keysym**.

state — A **mask16**.

Client Termination

14.5 The CLX functions affecting client termination are discussed in the following paragraphs.

When a display connection to an X server is closed, whether by an explicit call to **close-display** or by some external condition, the server automatically performs a sequence of operations to clean up server state information associated with the closed connection. The effect of these operations depends the *close-down mode* and the *save-set* that the client has specified for the closed display connection. The close-down mode of a display determines whether server resources allocated by the connection are freed or not. The save-set identifies windows that will remain after the connection is closed.

The display save-set is used primarily by window managers that reparent the top-level windows of other clients. For example, such a window manager can automatically create a frame window that encloses a top-level client window, along with a set of controls used for window management. Ordinarily, termination of the window manager client would then destroy all client windows! However, the window manager can prevent this by adding to its save-set those windows created by other clients that should be preserved.

When a display connection closes, an X server performs the following operations:

1. For each selection owned by a window created on the connection, the selection owner is set to **nil**.

2. An active or passive grab established for a window created on the connection is released.

3. If the connection has grabbed the server, the server is ungrabbed.

4. Server resources and colormap cells allocated by the connection are freed and destroyed, depending on the close-down mode, as follows:

   :retain-permanent — All resources are marked *permanent*, and no resources are destroyed. These resources can later be destroyed by a call to **kill-client**.

   :retain-temporary — All resources are marked *temporary*, and no resources are destroyed. These resources can later be destroyed by a call to **kill-client** or **kill-temporary-clients**.

   :destroy — All resources are destroyed.

When server resources allocated by a display connection are destroyed — whether by closing the connection with close-down mode **:destroy** or by later calling **kill-client** or **kill-temporary-clients** — then an X server performs the following operations on each member of the save-set before actually destroying resources.

1. If the save-set window is a descendant of a window created on the connection, the save-set window is reparented. The new parent is the closest ancestor such that the save-set window is no longer a descendant of any window created on the connection. The position of the reparented window with respect to its parent remains unchanged.

2. If the save-set window is unmapped, then it is mapped.

If the last connection open to an X server is closed with close-down mode **:destroy**, the server resets its state to restore all initial defaults. The server state after reset is the same as its initial state when first started. When an X server resets, it performs the following operations:

- All permanent and temporary server resources from previously-closed connections are destroyed.
- All but the predefined atoms are deleted.
- All root window properties are deleted.
- All device control attributes and mappings are restored to their original default values.
- The default background and cursor for all root windows are restored.
- The default font path is restored.
- The input focus is set to **:pointer-root**.
- The access control list is reset.

The following paragraphs describe the CLX functions used to:

- Add or remove a window from a display save-set.
- Return or change the display close-down mode.
- Force a connection to be closed or all its server resources to be destroyed.
- Force a connection to be closed and all temporary resources to be destroyed.

add-to-save-set *window* Function

Adds the specified *window* to the save-set of the *window* display. The *window* must have been created by some other display. Windows are removed automatically from the save-set when they are destroyed.

window — A **window**.

close-down-mode *display* Function

Returns:
mode — One of **:destroy**, **:retain-permanent**, or **:retain-temporary**.

Returns and (with **setf**) sets the close-down mode of the client's resources at connection close.

display — A **display**.

kill-client *display resource-id* Function

Closes the display connection which created the given *resource-id*. The *resource-id* must be valid, but need not belong to the given *display.*

If the closed connection was previously open, the connection is closed according to its close-down mode. Otherwise, if the connection had been previously terminated with close-down mode **:retain-permanent** or **:retain-temporary**, then all its retained server resources — both permanent and temporary — are destroyed.

display — A **display**.

resource-id — A valid **card29** resource ID.

kill-temporary-clients *display* Function

Closes the *display* connection and destroys all retained temporary server resources for this and all previously-terminated connections.

If the *display* connection was previously open, the connection is closed according to its close-down mode. Otherwise, if the *display* connection had been previously terminated with close-down mode **:retain-permanent** or **:retain-temporary**, then all its retained server resources — both permanent and temporary — are destroyed.

display — A **display**.

remove-from-save-set *window* Function

Removes the specified *window* from the save-set of the *window* display. The *window* must have been created by some other display. Windows are removed automatically from the save-set when they are destroyed.

window — A **window**.

Managing Host Access

14.6 An X server maintains a list of hosts from which client programs can be run. Only clients executing on hosts that belong to this *access control list* are allowed to open a connection to the server. Typically, the access control list can be changed by clients running on the same host as the server. Some server implementations can also implement other authorization mechanisms in addition to, or in place of, this mechanism. The action of this mechanism can be conditional based on the authorization protocol name and data received by the server at connection setup.

The following paragraphs describe the CLX functions used to:

- Add or remove hosts on the access control list.
- Return the hosts on the access control list.
- Return or change the state of the access control list mechanism

access-control *display* Function

Returns:
enabled-p — Type **boolean**.

Returns and (with **setf**) changes the state of the access control list mechanism for the *display* server. Returns true if access control is enabled; otherwise, **nil** is returned. If enabled, the access control list is used to validate each client during connection setup.

Only a client running on the same host as the server is allowed to enable or disable the access control list mechanism.

display — A **display**.

access-hosts *display* &key (**:result-type 'list**) Function

Returns:
hosts — **sequence** of **string**.
enabled-p — Type **boolean**.

Returns a sequence containing the *hosts* that belong to the access control list of the *display* server. Elements of the returned *hosts* sequence are either strings or some other type of object recognized as a host name by **add-access-host** and **remove-access-host**. The second returned value specifies whether the access control list mechanism is currently enabled or disabled (see **access-control**).

display — A **display**.

:result-type — The type of hosts sequence to return.

add-access-host *display host* Function

Adds the specified *host* to the access control list. Only a client running on the same host as the server can change the access control list.

display — A **display**.

host — A host name. Either a string or some other implementation-dependent type.

remove-access-host *display host* Function

Removes the specified *host* from the access control list. Only a client running on the same host as the server can change the access control list.

display — A **display**.

host — A host name. Either a string or some other implementation-dependent type.

Screen Saver

14.7 To prevent monitor damage, an X server implements a screen saver function which blanks screens during periods of unuse. The screen saver can be in one of three states:

- Disabled — No screen blanking is done and screen content remains unchanged.
- Deactivated — The server is being used. When the server input devices are unused for a specific amount of time, the screen saver becomes activated.
- Activated — The server input devices are unused. The screen saver blanks all server screens or displays a server-dependent image. As soon as an input event from either the pointer or the keyboard occurs, the screen saver is deactivated and its timer is reset.

The following paragraphs describe the CLX functions used to:

- Return or change screen saver control values.
- Activate or reset the screen saver

activate-screen-saver *display* Function

Activates the screen saver for the *display* server.

display — A **display**.

reset-screen-saver *display* Function

Deactivates the screen saver for the *display* server (if necessary) and resets its timer, just as if a pointer or keyboard event had occurred.

display — A **display**.

screen-saver *display* Function

Returns:
timeout, *period* — Type **int16**.
blanking, *exposures* — One of **:yes** or **:no**.

Returns the current control values for the *display* server screen saver. See **set-screen-saver**.

display — A **display**.

set-screen-saver *display timeout period blanking exposures* Function

Changes the current control values for the *display* server screen saver. The screen saver is reset. The screen saver is also disabled if:

- *timeout* is zero, or
- Both *blanking* and *exposures* are disabled and the server cannot regenerate the screen contents without sending **:exposure** events.

The *timeout* specifies the (non-negative) number of seconds of input device inactivity that must elapse before the screen saver is activated. The *timeout* can be set to **:default** to restore the server default timeout interval.

If *blanking* is **:yes** and the screen hardware supports blanking, blanking is enabled; that is, the screen saver will simply blank all screens when it is activated. *blanking* can be set to **:default** to restore the server default state for blanking.

If *exposures* is **:yes**, exposures are enabled. If exposures are enabled, or if the server is capable of regenerating screen contents without sending **:exposure** events, the screen saver will display some server-dependent image when activated. Frequently, this image will consist of a repeating animation sequence, in which case *period* specifies the (non-negative) number of seconds for each repetition. A *period* of zero is a hint that no repetition should occur.

display — A **display**.

timeout — Specifies the delay until timeout takes over.

period — Specifies the periodic change interval, if used.

blanking — Specifies whether the blanking option is available.

exposures — Specifies whether exposures are allowed during blanking.

EXTENSIONS

15

Extensions

15.1 The X Window System is based on a core protocol which can be extended to provide new functionality. An extension is generally represented by an additional set of requests or event types that are implemented by an X server supporting the extension. By definition, a client program using an extension may not be portable to other servers. However, extensions allow different server implementations and different sites to add their own special features to X, without disrupting clients that rely only on the core protocol.

Extensions are identified by assigning them unique name strings and major protocol numbers. A client program can request an X server to use a protocol extension by furnishing the extension protocol number as an argument to **open-display**. The X Consortium maintains a registry of standard extension names and protocol numbers.

The following paragraphs describe the CLX functions used to:

- List all supported extensions.
- Find out if a given extension is supported.

list-extensions *display* &key (**:result-type 'list**) Function

Returns:
names — Type **sequence** of **string**.

Returns a sequence containing the *names* of all extensions supported by the *display* server.

display — A **display**.

:result-type — The type of name sequence to return.

query-extension *display name* Function

Returns:
major-opcode, *first-event*, *first-error* — Type **card8** or **null**.

Returns the *major-opcode* for the given extension *name* support by the *display* server. If the extension is not supported, only **nil** values are returned. The extension *name* must contain only ISO Latin-1 characters; case is significant.

If the extension involves additional event types, the *first-event* returned is the base event type code for new events; otherwise, the *first-event* is **nil**. If the extension involves additional error codes, the *first-error* returned is the base code for new errors; otherwise, the *first-error* is **nil**. The formats of error and event messages sent by the server are completely defined by the extension.

display — A **display**.

name — An extension name string.

ERRORS 16

Introduction

16.1 CLX error conditions are hierarchial. The base error condition is **x-error**, and all other conditions are built on top of **x-error**. **x-error** can be built on a lower-level condition that is implementation dependent (this is probably the **error** condition).

define-condition *name* (*parent-types**) [({*slot-specifier**}) {*option**}] Macro

Any new condition type must be defined with the **define-condition** macro. A condition type has a name, parent types, report message, and any number of slot items. See the *Lisp Reference* manual for further information regarding **define-condition**.

The following are the predefined error conditions that can occur in CLX.

access-error Condition

An **access-error** can occur for several reasons:

- A client attempted to grab a key/button combination already grabbed by another client
- A client attempted to free a colormap entry that it did not already allocate
- A client attempted to store into a read-only colormap entry
- A client attempted to modify the access control list from other than the local (or otherwise authorized) host
- A client attempted to select an event type that another client has already selected, and, that at most, one client can select at a time

An **access-error** is a special case of the more general **request-error** (see page 16-189 for information on **request-error**).

alloc-error Condition

The server failed to allocate the requested resource or server memory.

An **alloc-error** is a special case of the more general **request-error** (see page 16-189 for information on **request-error**).

atom-error Condition

A value for an *atom* argument does not name a defined atom.

An **atom-error** is a special case of the more general **request-error** (see page 16-189 for information on **request-error**).

closed-display Condition

The **closed-display** condition is signaled when trying to read or write a closed display (that is, **close-display** has been called on the **display** object, or a server-disconnect occurred). The **closed-display** object is reported with the error.

A **closed-display** condition is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

colormap-error Condition

A value for a *colormap* argument does not name a defined colormap.

A **colormap-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

connection-failure Condition

Signaled when an X11 server refuses a connection. The following items are reported along with the error:

- *major-version* — The major version of the X server code.
- *minor-version* — The minor version of the X server code.
- *host* — The host name for the X server.
- *display* — The display on which the error occurred.
- *reason* — A string indicating why the connection failed.

A **connection-failure** is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

cursor-error Condition

A value for a *cursor* argument does not name a defined cursor.

A **cursor-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

device-busy Condition

Signaled by (**setf** (**pointer-mapping** *display*) **mapping**) when the **set-pointer-mapping** request returns a busy status. A similar condition occurs in **set-modifier-mapping**, but in this case, it returns a boolean indicating success, rather than signaling an error. The **device-busy** condition returns the display object as part of the error.

A **device-busy** condition is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

drawable-error Condition

A value for a *drawable* argument does not name a defined window or pixmap.

A **drawable-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

font-error Condition

A value for a *font* or *gcontext* argument does not name a defined font.

A **font-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

gcontext-error Condition

A value for a *gcontext* argument does not name a defined GContext.

A **gcontext-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

id-choice-error Condition

The value chosen for a resource identifier is either not included in the range assigned to the client or is already in use. Under normal circumstances, this cannot occur and should be considered a server or CLX library error.

An **id-choice-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

implementation-error Condition

The server does not implement some aspect of the request. A server that generates this error for a core request is deficient. As such, this error is not listed for any of the requests. However, clients should be prepared to receive such errors and either handle or discard them.

An **implementation-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

length-error Condition

The length of a request is shorter or longer than that minimally required to contain the arguments. This usually means an internal CLX error.

A **length-error** is a special case of the more general **resource-error** (see page 16-189 for information on **resource-error**).

lookup-error Condition

CLX has the option of caching different resource types (see ***clx-cached-types***) in a hash table by resource ID. When looking up an object in the hash table, if the type of the object is wrong, a **lookup-error** is signaled.

For example: The cursor with ID 123 is interned in the hash table. An event is received with a field for window 123. When 123 is looked up in the hash table, a cursor is found. Since a window was expected, a **lookup-error** is signaled. This error indicates a problem with the extension code being used. The following items are reported along with the error:

- *id* — The resource ID.
- *display* — The display being used.
- *type* — The resource type.
- *object* — The **resource** object.

A **lookup-error** is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

match-error Condition

In a graphics request, the root and depth of the GContext does not match that of the drawable. An **:input-only** window is used as a drawable. Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request. An **:input-only** window locks this attribute. The values do not exist for an **:input-only** window.

A **match-error** is a special case of the more general **request-error** (see page 16-189 for information on **request-error**).

missing-parameter Condition

One or more of the required keyword parameters is missing or **nil**. The missing parameters are reported along with the error.

A **missing-parameter** condition is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

name-error Condition

A font or color of the specified name does not exist.

A **name-error** is a special case of the more general **request-error** (see page 16-189 for information on **request-error**).

pixmap-error Condition

A value for a *pixmap* argument does not name a defined pixmap.

A **pixmap-error** is a special case of the more general **resource-error**. (See page 16-189 for information on **resource-error**.)

reply-length-error (**x-error**) (*slots**) Condition

The reply to a request has an unexpected length. The following items are reported along with the error:

- *reply-length* — The actual reply length.
- *expected-length* — The expected reply length.
- *display* — The display on which the error occurred.

A **reply-length-error** is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

reply-timeout Condition

The ***reply-timeout*** parameter specifies the maximum number of seconds to wait for a request reply, or **nil** to wait forever (the default). When a reply has not been received after ***reply-timeout*** seconds, the **reply-timeout** condition is signaled. The *timeout period* and *display* are reported along with the error.

A **reply-timeout** condition is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

request-error Condition

The following items are reported along with the error:

The major or minor opcode does not specify a valid request.

- *display* — The display on which the error occurred.
- *error-key* — The error (sub)type.
- *major* — The major opcode.
- *minor* — The minor opcode.
- *sequence* — The actual sequence number.
- *current-sequence* — The current sequence number.

A **request-error** condition is a special case of the more general **x-error** (see page 16-190 for information on **x-error**).

resource-error Condition

All X11 errors for incorrect resource IDs are built on top of **resource-error**. These are **colormap-error**, **cursor-error**, **drawable-error**, **font-error**, **gcontext-error**, **id-choice-error**, **pixmap-error** and **window-error**. **resource-error** is never signaled directly.

A **resource-error** is a special case of the more general **request-error** (see page 16-189 for information on **request-error**).

sequence-error Condition

All X11 request replies contain the sequence number of their request. If a reply's sequence does not match the request count, a **sequence-error** is signaled. A **sequence-error** usually indicates a locking problem with a multi-processing Lisp. The following items are reported along with the error:

- *display* — The display on which the error occurred.
- *req-sequence* — The sequence number in the reply.
- *msg-sequence* — The current sequence number.

A **sequence-error** condition is a special case of the more general **x-error**. (See page 16-190 for information on **x-error**.)

server-disconnect Condition

The connection to the server was lost. The display on which the error occurred is reported along with the error.

A **server-disconnect** condition is a special case of the more general **x-error**. (See page 16-190 for information on **x-error**.)

unexpected-reply Condition

A reply was found when none was expected. This indicates a problem with the extension code. The following items are reported along with the error:

- *display* — The display on which the error occurred.
- *req-sequence* — The sequence number in the reply.
- *msg-sequence* — The current sequence number.
- *length* — The message length of the reply.

An **unexpected-reply** condition is a special case of the more general **x-error**. (See page 16-190 for information on **x-error**.)

unknown-error (**request-error**) (**error-code**) Condition

An error was received from the server with an unknown error code. This indicates a problem with the extension code. The undefined error code is reported.

An **unknown-error** is a special case of the more general **request-error**. (See page 16-189 for information on **request-error**.)

value-error (**request-error**) (**value**) Condition

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error. The erroneous value is reported.

A **value-error** is a special case of the more general **request-error**. (See page 16-189 for information on **request-error**.)

window-error (**resource-error**) Condition

A value for a *window* argument does not name a defined window.

A **window-error** is a special case of the more general **resource-error**. (See page 16-189 for information on **resource-error**.)

x-error Condition

This is the most general error condition upon which all other conditions are defined.

PROTOCOL VS. CLX FUNCTIONAL CROSS-REFERENCE LISTING

A

X11 Request Name	CLX Function Name
AllocColor	**alloc-color**
AllocColorCells	**alloc-color-cells**
AllocColorPlanes	**alloc-color-planes**
AllocNamedColor	**alloc-color**
AllowEvents	**allow-events**
Bell	**bell**
ChangeAccessControl	(**setf** (**access-control** *display*)
ChangeActivePointerGrab	**change-active-pointer-grab**
ChangeCloseDownMode	(**setf** (**close-down-mode** *display*))
ChangeGC	**force-gcontext-changes**
	(See **with-gcontext**)
	(**setf** (**gcontext-function** *gc*))
	(**setf** (**gcontext-plane-mask** *gc*))
	(**setf** (**gcontext-foreground** *gc*))
	(**setf** (**gcontext-background** *gc*))
	(**setf** (**gcontext-line-width** *gc*))
	(**setf** (**gcontext-line-style** *gc*))
	(**setf** (**gcontext-cap-style** *gc*))
	(**setf** (**gcontext-join-style** *gc*))
	(**setf** (**gcontext-fill-style** *gc*))
	(**setf** (**gcontext-fill-rule** *gc*))
	(**setf** (**gcontext-tile** *gc*))
	(**setf** (**gcontext-stipple** *gc*))
	(**setf** (**gcontext-ts-x** *gc*))
	(**setf** (**gcontext-ts-y** *gc*))
	(**setf** (**gcontext-font** *gc* &optional **metrics-p**))
	(**setf** (**gcontext-subwindow-mode** *gc*))
	(**setf** (**gcontext-exposures** *gc*)))
	(**setf** (**gcontext-clip-x** *gc*))
	(**setf** (**gcontext-clip-y** *gc*))
	(**setf** (**gcontext-clip-mask** *gc* &optional **ordering**))
	(**setf** (**gcontext-dash-offset** *gc*))
	(**setf** (**gcontext-dashes** *gc*))
	(**setf** (**gcontext-arc-mode** *gc*))
	(**setf** (**gcontext-clip-ordering** *gc*))

X11 Request Name	CLX Function Name
ChangeHosts	**add-access-host**
ChangeHosts	**remove-access-host**
ChangeKeyboardControl	**change-keyboard-control**
ChangePointerControl	**change-pointer-control**
ChangeProperty	**change-property**
ChangeSaveSet	**remove-from-save-set**
ChangeSaveSet	**add-to-save-set**
ChangeWindowAttributes	(See **with-state**)
	(**setf** (**window-background** *window*))
	(**setf** (**window-border** *window*))
	(**setf** (**window-bit-gravity** *window*))
	(**setf** (**window-gravity** *window*))
	(**setf** (**window-backing-store** *window*))
	(**setf** (**window-backing-planes** *window*))
	(**setf** (**window-backing-pixel** *window*))
	(**setf** (**window-override-redirect** *window*)
	(setf (window-save-under *window***))**
	(**setf** (**window-colormap** *window*))
	(**setf** (**window-cursor** *window*))
	(**setf** (**window-event-mask** *window*))
	(**setf** (**window-do-not-propagate-mask** *window*))
CirculateWindow	**circulate-window-down**
CirculateWindow	**circulate-window-up**
ClearToBackground	**clear-area**
CloseFont	**close-font**
ConfigureWindow	(See **with-state**)
	(**setf** (**drawable-x** *drawable*))
	(**setf** (**drawable-y** *drawabl*e))
	(**setf** (**drawable-width** *drawable*))
	(**setf** (**drawable-height** *drawable*))
	(**setf** (**drawable-depth** *drawable*))
	(**setf** (**drawable-border-width** *drawable*))
	(**setf** (**window-priority** *window* &optional **sibling**))
ConvertSelection	**convert-selection**
CopyArea	**copy-area**
CopyColormapAndFree	**copy-colormap-and-free**
CopyGC	**copy-gcontext**
CopyGC	**copy-gcontext-components**
CopyPlane	**copy-plane**
CreateColormap	**create-colormap**
CreateCursor	**create-cursor**
CreateGC	**create-gcontext**
CreateGlyphCursor	**create-glyph-cursor**
CreatePixmap	**create-pixmap**
CreateWindow	**create-window**
DeleteProperty	**delete-property**
DestroySubwindows	**destroy-subwindows**
DestroyWindow	**destroy-window**
FillPoly	**draw-lines**
ForceScreenSaver	**reset-screen-saver**
ForceScreenSaver	**activate-screen-saver**
FreeColormap	**free-colormap**
FreeColors	**free-colors**
FreeCursor	**free-cursor**

X11 Request Name	CLX Function Name
FreeGC	**free-gcontext**
FreePixmap	**free-pixmap**
GetAtomName	**atom-name**
GetFontPath	**font-path**
GetGeometry	(See **with-state**)
	drawable-root
	drawable-x
	drawable-y
	drawable-width
	drawable-height
	drawable-depth
	drawable-border-width
GetImage	**get-raw-image**
GetInputFocus	**input-focus**
GetKeyboardControl	**keyboard-control**
GetKeyboardMapping	**keyboard-mapping**
GetModifierMapping	**modifier-mapping**
GetMotionEvents	**motion-events**
GetPointerControl	**pointer-control**
GetPointerMapping	**pointer-mapping**
GetProperty	**get-property**
GetScreenSaver	**screen-saver**
GetSelectionOwner	**selection-owner**
GetWindowAttributes	(See **with-state**)
	window-visual
	window-class
	window-bit-gravity
	window-gravity
	window-backing-store
	window-backing-planes
	window-backing-pixel
	window-save-under
	window-override-redirect
	window-event-mask
	window-do-not-propagate-mask
	window-colormap
	window-colormap-installed-p
	window-all-event-masks
	window-map-state
GrabButton	**grab-button**
GrabKey	**grab-key**
GrabKeyboard	**grab-keyboard**
GrabPointer	**grab-pointer**
GrabServer	**grab-server**
ImageText16	**draw-image-glyphs**
ImageText16	**draw-image-glyph**
ImageText8	**draw-image-glyphs**
InstallColormap	**install-colormap**
InternAtom	**find-atom**
InternAtom	**intern-atom**
KillClient	**kill-temporary-clients**
KillClient	**kill-client**
ListExtensions	**list-extensions**
ListFonts	**list-font-names**
ListFontsWithInfo	**list-fonts**
ListHosts	**access-control**

X11 Request Name	CLX Function Name
ListHosts	**access-hosts**
ListInstalledColormaps	**installed-colormaps**
ListProperties	**list-properties**
LookupColor	**lookup-color**
MapSubwindows	**map-subwindows**
MapWindow	**map-window**
OpenFont	**open-font**
PolyArc	**draw-arc**
PolyArc	**draw-arcs**
PolyFillArc	**draw-arc**
PolyFillArc	**draw-arcs**
PolyFillRectangle	**draw-rectangle**
PolyFillRectangle	**draw-rectangles**
PolyLine	**draw-line**
PolyLine	**draw-lines**
PolyPoint	**draw-point**
PolyPoint	**draw-points**
PolyRectangle	**draw-rectangle**
PolyRectangle	**draw-rectangles**
PolySegment	**draw-segments**
PolyText16	**draw-glyph**
PolyText16	**draw-glyphs**
PolyText8	**draw-glyphs**
PutImage	**put-raw-image**
QueryBestSize	**query-best-cursor**
QueryBestSize	**query-best-stipple**
QueryBestSize	**query-best-tile**
QueryColors	**query-colors**
QueryExtension	**query-extension**
QueryFont	**font-name**
	font-name
	font-direction
	font-min-char
	font-max-char
	font-min-byte1
	font-max-byte1
	font-min-byte2
	font-max-byte2
	font-all-chars-exist-p
	font-default-char
	font-ascent
	font-descent
	font-properties
	font-property
	char-left-bearing
	char-right-bearing
	char-width
	char-ascent
	char-descent
	char-attributes
	min-char-left-bearing
	min-char-right-bearing
	min-char-width
	min-char-ascent
	min-char-descent
	min-char-attributes

X11 Request Name	CLX Function Name
	max-char-left-bearing
	max-char-right-bearing
	max-char-width
	max-char-ascent
	max-char-descent
	max-char-attributes
QueryKeymap	**query-keymap**
QueryPointer	**global-pointer-position**
QueryPointer	**pointer-position**
QueryPointer	**query-pointer**
QueryTextExtents	**text-extents**
QueryTextExtents	**text-width**
QueryTree	**query-tree**
RecolorCursor	**recolor-cursor**
ReparentWindow	**reparent-window**
RotateProperties	**rotate-properties**
SendEvent	**send-event**
SetClipRectangles	**force-gcontext-changes**
	(See **with-gcontext**)
	(**setf** (**gcontext-clip-x** *gc*))
	(**setf** (**gcontext-clip-y** *gc*))
	(**setf** (**gcontext-clip-mask** *gc* &optional **ordering**))
	(**setf** (**gcontext-clip-ordering** *gc*))
SetDashes	**force-gcontext-changes**
	(See **with-gcontext**)
	(**setf** (**gcontext-dash-offset** *gc*))
	(**setf** (**gcontext-dashes** *gc*))
SetFontPath	(**setf** (**font-path** *font*)
SetInputFocus	**set-input-focus**
SetKeyboardMapping	**change-keyboard-mapping**
SetModifierMapping	**set-modifier-mapping**
SetPointerMapping	**set-pointer-mapping**
SetScreenSaver	**set-screen-saver**
SetSelectionOwner	**set-selection-owner**
StoreColors	**store-color**
StoreColors	**store-colors**
StoreNamedColor	**store-color**
StoreNamedColor	**store-colors**
TranslateCoords	**translate-coordinates**
UngrabButton	**ungrab-button**
UngrabKey	**ungrab-key**
UngrabKeyboard	**ungrab-keyboard**
UngrabPointer	**ungrab-pointer**
UngrabServer	**ungrab-server**
UninstallColormap	**uninstall-colormap**
UnmapSubwindows	**unmap-subwindows**
UnmapWindow	**unmap-window**
WarpPointer	**warp-pointer**
WarpPointer	**warp-pointer-if-inside**
WarpPointer	**warp-pointer-relative**
WarpPointer	**warp-pointer-relative-if-inside**
ListHosts	**access-control**
ListHosts	**access-hosts**
ForceScreenSaver	**activate-screen-saver**
ChangeHosts	**add-access-host**

X11 Request Name	CLX Function Name
ChangeSaveSet	**add-to-save-set**
AllocColor	**alloc-color**
AllocNamedColor	**alloc-color**
AllocColorCells	**alloc-color-cells**
AllocColorPlanes	**alloc-color-planes**
AllowEvents	**allow-events**
GetAtomName	**atom-name**
Bell	**bell**
ChangeActivePointerGrab	**change-active-pointer-grab**
ChangeKeyboardControl	**change-keyboard-control**
SetKeyboardMapping	**change-keyboard-mapping**
ChangePointerControl	**change-pointer-control**
ChangeProperty	**change-property**
QueryFont	**char-ascent**
QueryFont	**char-attributes**
QueryFont	**char-descent**
QueryFont	**char-left-bearing**
QueryFont	**char-right-bearing**
QueryFont	**char-width**
CirculateWindow	**circulate-window-down**
CirculateWindow	**circulate-window-up**
ClearToBackground	**clear-area**
CloseFont	**close-font**
ConvertSelection	**convert-selection**
CopyArea	**copy-area**
CopyColormapAndFree	**copy-colormap-and-free**
CopyGC	**copy-gcontext**
CopyGC	**copy-gcontext-components**
CopyPlane	**copy-plane**
CreateColormap	**create-colormap**
CreateCursor	**create-cursor**
CreateGC	**create-gcontext**
CreateGlyphCursor	**create-glyph-cursor**
CreatePixmap	**create-pixmap**
CreateWindow	**create-window**
DeleteProperty	**delete-property**
DestroySubwindows	**destroy-subwindows**
DestroyWindow	**destroy-window**
PolyArc	**draw-arc**
PolyArc	**draw-arcs**
PolyText16	**draw-glyph**
PolyText16	**draw-glyphs**
PolyText8	**draw-glyphs**
ImageText16	**draw-image-glyph**
ImageText16	**draw-image-glyphs**
ImageText8	**draw-image-glyphs**
PolyLine	**draw-line**
PolyLine	**draw-lines**
PolyPoint	**draw-point**
PolyPoint	**draw-points**
PolyFillRectangle	**draw-rectangle**
PolyRectangle	**draw-rectangle**
PolyFillRectangle	**draw-rectangles**
PolyRectangle	**draw-rectangles**
PolySegment	**draw-segments**
GetGeometry	**drawable-border-width**

X11 Request Name	CLX Function Name
GetGeometry	**drawable-depth**
GetGeometry	**drawable-height**
GetGeometry	**drawable-root**
GetGeometry	**drawable-width**
GetGeometry	**drawable-x**
GetGeometry	**drawable-y**
FillPoly	**fill-polygon**
InternAtom	**find-atom**
QueryFont	**font-all-chars-exist-p**
QueryFont	**font-ascent**
QueryFont	**font-default-char**
QueryFont	**font-descent**
QueryFont	**font-direction**
QueryFont	**font-max-byte1**
QueryFont	**font-max-byte2**
QueryFont	**font-max-char**
QueryFont	**font-min-byte1**
QueryFont	**font-min-byte2**
QueryFont	**font-min-char**
QueryFont	**font-name**
QueryFont	**font-name**
GetFontPath	**font-path**
QueryFont	**font-properties**
QueryFont	**font-property**
ChangeGC	**force-gcontext-changes**
SetClipRectangles	**force-gcontext-changes**
SetDashes	**force-gcontext-changes**
FreeColormap	**free-colormap**
FreeColors	**free-colors**
FreeCursor	**free-cursor**
FreeGC	**free-gcontext**
FreePixmap	**free-pixmap**
GetProperty	**get-property**
GetImage	**get-raw-image**
QueryPointer	**global-pointer-position**
GrabButton	**grab-button**
GrabKey	**grab-key**
GrabKeyboard	**grab-keyboard**
GrabPointer	**grab-pointer**
GrabServer	**grab-server**
GrabServer	**with-server-grabbed**
GetInputFocus	**input-focus**
InstallColormap	**install-colormap**
ListInstalledColormaps	**installed-colormaps**
InternAtom	**intern-atom**
GetKeyboardControl	**keyboard-control**
GetKeyboardMapping	**keyboard-mapping**
KillClient	**kill-client**
KillClient	**kill-temporary-clients**
ListExtensions	**list-extensions**
ListFonts	**list-font-names**
ListFontsWithInfo	**list-fonts**
ListProperties	**list-properties**
LookupColor	**lookup-color**
MapSubwindows	**map-subwindows**
MapWindow	**map-window**

X11 Request Name	CLX Function Name
QueryFont	**max-char-ascent**
QueryFont	**max-char-attributes**
QueryFont	**max-char-descent**
QueryFont	**max-char-left-bearing**
QueryFont	**max-char-right-bearing**
QueryFont	**max-char-width**
QueryFont	**min-char-ascent**
QueryFont	**min-char-attributes**
QueryFont	**min-char-descent**
QueryFont	**min-char-left-bearing**
QueryFont	**min-char-right-bearing**
QueryFont	**min-char-width**
GetModifierMapping	**modifier-mapping**
GetMotionEvents	**motion-events**
OpenFont	**open-font**
GetPointerControl	**pointer-control**
GetPointerMapping	**pointer-mapping**
QueryPointer	**pointer-position**
PutImage	**put-raw-image**
QueryBestSize	**query-best-cursor**
QueryBestSize	**query-best-stipple**
QueryBestSize	**query-best-tile**
QueryColors	**query-colors**
QueryExtension	**query-extension**
QueryKeymap	**query-keymap**
QueryPointer	**query-pointer**
QueryTree	**query-tree**
RecolorCursor	**recolor-cursor**
ChangeHosts	**remove-access-host**
ChangeSaveSet	**remove-from-save-set**
ReparentWindow	**reparent-window**
ForceScreenSaver	**reset-screen-saver**
RotateProperties	**rotate-properties**
GetScreenSaver	**screen-saver**
GetSelectionOwner	**selection-owner**
SendEvent	**send-event**
ChangeAccessControl	**set-access-control**
ChangeCloseDownMode	**set-close-down-mode**
SetInputFocus	**set-input-focus**
SetModifierMapping	**set-modifier-mapping**
SetPointerMapping	**set-pointer-mapping**
SetScreenSaver	**set-screen-saver**
SetSelectionOwner	**set-selection-owner**
StoreColors	**store-color**
StoreColors	**store-colors**
StoreNamedColor	**store-color**
StoreNamedColor	**store-colors**
QueryTextExtents	**text-extents**
QueryTextExtents	**text-width**
TranslateCoords	**translate-coordinates**
UngrabButton	**ungrab-button**
UngrabKey	**ungrab-key**
UngrabKeyboard	**ungrab-keyboard**
UngrabPointer	**ungrab-pointer**
UngrabServer	**ungrab-server**
UngrabServer	**with-server-grabbed**

X11 Request Name	**CLX Function Name**
UninstallColormap	**uninstall-colormap**
UnmapSubwindows	**unmap-subwindows**
UnmapWindow	**unmap-window**
WarpPointer	**warp-pointer**
WarpPointer	**warp-pointer-if-inside**
WarpPointer	**warp-pointer-relative**
WarpPointer	**warp-pointer-relative-if-inside**
GetWindowAttributes	**window-all-event-masks**
GetWindowAttributes	**window-backing-pixel**
GetWindowAttributes	**window-backing-planes**
GetWindowAttributes	**window-backing-store**
GetWindowAttributes	**window-bit-gravity**
GetWindowAttributes	**window-class**
GetWindowAttributes	**window-colormap**
GetWindowAttributes	**window-colormap-installed-p**
GetWindowAttributes	**window-do-not-propagate-mask**
GetWindowAttributes	**window-event-mask**
GetWindowAttributes	**window-gravity**
GetWindowAttributes	**window-map-state**
GetWindowAttributes	**window-override-redirect**
GetWindowAttributes	**window-save-under**
GetWindowAttributes	**window-visual**
ConfigureWindow	(**setf** (**drawable-border-width** *drawable*))
ConfigureWindow	(**setf** (**drawable-depth** *drawable*))
ConfigureWindow	(**setf** (**drawable-height** *drawable*))
ConfigureWindow	(**setf** (**drawable-width** *drawabl*e))
ConfigureWindow	(**setf** (**drawable-x** *drawable*))
ConfigureWindow	(**setf** (**drawable-y** *drawable*))
SetFontPath	(**setf** (**font-path** *font*) **paths**)
ChangeGC	(**setf** (**gcontext-arc-mode** *gc*))
ChangeGC	(**setf** (**gcontext-background** *gc*))
ChangeGC	(**setf** (**gcontext-cap-style** *gc*))
SetClipRectangles	(**setf** (**gcontext-clip-mask** *gc* &optional **ordering**))
SetClipRectangles	(**setf** (**gcontext-clip-ordering** *gc*))
SetClipRectangles	(**setf** (**gcontext-clip-x** *gc*))
SetClipRectangles	(**setf** (**gcontext-clip-y** *gc*))
SetDashes	(**setf** (**gcontext-dash-offset** *gc*))
SetDashes	(**setf** (**gcontext-dashes** *gc*))
ChangeGC	(**setf** (**gcontext-exposures** *gc*))
ChangeGC	(**setf** (**gcontext-fill-rule** *gc*) **keyword**)
ChangeGC	(**setf** (**gcontext-fill-style** *gc*) **keyword**)
ChangeGC	(**setf** (**gcontext-font** *gc* &optional **metrics-p**)
ChangeGC	(**setf** (**gcontext-foreground** *gc*) **card32**)
ChangeGC	(**setf** (**gcontext-function** *gc*))
ChangeGC	(**setf** (**gcontext-join-style** *gc*) **keyword**)
ChangeGC	(**setf** (**gcontext-line-style** *gc*) **keyword**)
ChangeGC	(**setf** (**gcontext-line-width** *gc*) **card16**)
ChangeGC	(**setf** (**gcontext-plane-mask** *gc*) **card32**)
ChangeGC	(**setf** (**gcontext-stipple** *gc*) **pixmap**)
ChangeGC	(**setf** (**gcontext-subwindow-mode** *gc*))
ChangeGC	(**setf** (**gcontext-tile** *gc*))
ChangeGC	(**setf** (**gcontext-ts-x** *gc*))
ChangeGC	(**setf** (**gcontext-ts-y** *gc*))
ChangeWindowAttributes	(**setf** (**window-background** *window*))

X11 Request Name	**CLX Function Name**
ChangeWindowAttributes	(**setf** (**window-backing-pixel** *window*))
ChangeWindowAttributes	(**setf** (**window-backing-planes** *window*))
ChangeWindowAttributes	(**setf** (**window-backing-store** *window*))
ChangeWindowAttributes	(**setf** (**window-bit-gravity** *window*))
ChangeWindowAttributes	(**setf** (**window-border** *window*))
ChangeWindowAttributes	(**setf** (**window-colormap** *window*))
ChangeWindowAttributes	(**setf** (**window-cursor** *window*))
ChangeWindowAttributes	(**setf** (**window-do-not-propagate-mask** *window*))
ChangeWindowAttributes	(**setf** (**window-event-mask** *window*))
ChangeWindowAttributes	(**setf** (**window-gravity** *window*))
ChangeWindowAttributes	(**setf** (**window-override-redirect** *window*))
ConfigureWindow	(**setf** (**window-priority** *window* &optional **sibling**))
ChangeWindowAttributes	(**setf** (**window-save-under** *window*))

GLOSSARY

a

access control list X maintains a list of hosts from which client programs can be run. By default, only programs on the local host can use the display, plus any hosts specified in an initial list read by the server. This *access control list* can be changed by clients on the local host. Some server implementations can also implement other authorization mechanisms in addition to or in place of this mechanism. The action of this mechanism can be conditional based on the authorization protocol name and data received by the server at connection setup.

action A function that is designed to handle an input event. CLUE input processing consists of matching an event with an event specification found in a contact's **event-translations** slot and then calling actions associated with the matching event specification.

active grab A grab is *active* when the pointer or keyboard is actually owned by the single grabbing client.

ancestors If W is an inferior of A, then A is an *ancestor* of W.

atom A unique ID corresponding to a string name. Atoms are used to identify properties, types, and selections.

b

backing store When a server maintains the contents of a window, the off-screen saved pixels are known as a *backing store*.

before action An action of a **contact-display** that is called when an event is dispatched to a contact, but before any other contact input processing is performed.

bit gravity When a window is resized, the contents of the window are not necessarily discarded. It is possible to request the server to relocate the previous contents to some region of the window. This attraction of window contents for some location of a window is known as *bit gravity.*

bitmap A pixmap of depth one.

button grabbing Buttons on the pointer can be passively *grabbed* by a client. When the button is pressed, the pointer is then actively grabbed by the client.

byte order For image (pixmap/bitmap) data, byte order is defined by the server, and clients with different native byte ordering must swap bytes as necessary. For all other parts of the protocol, the byte order is defined by the client, and the server swaps bytes as necessary.

c

callback A function that represents a connection between a contact and the rest of an application program. A contact calls a callback function in order to report the results of the user interface component that it represents.

children First-level subwindows of a window.

class event translations Event translations that belong to all instances of a contact class. A class event translation is created by the **defevent** macro.

class resources Resources defined for each instance of a contact class. Also see constraint resources.

click A **:button-press** event followed immediately by a **:button-release** event for the same button, with no intervening change in pointer position or modifier key state.

client An application program connects to the window system server by some interprocess communication (IPC) path, such as a TCP connection or a shared memory buffer. This program is referred to as a *client* of the window system server. More precisely, the client is the IPC path itself. A program with multiple paths open to the server is viewed as multiple clients by the protocol. Resource lifetimes are controlled by connection lifetimes, not by program lifetimes.

clipping regions In a graphics context, a bitmap or list of rectangles can be specified to restrict output to a particular region of the window. The image defined by the bitmap or rectangles is called a *clipping region*.

colormap A set of entries defining color values. The colormap associated with a window is used to display the contents of the window. Each pixel value indexes the colormap to produce RGB values that drive the guns of a monitor. Depending on hardware limitations, one or more colormaps can be installed at one time, such that windows associated with those maps display with correct colors.

composite A subclass of **contact** representing contacts that are the parents of other contacts. A composite provides geometry management and input focus management services for the contacts that are its children.

complete resource class A list of symbols containing the class of the contact, the class of the contact's **parent** (and so on), and the class of the **contact-display** to which the contact belongs. The complete resource class is one of the two items used as a key by a CLUE application in order to access a contact resource value in a resource database.

complete resource name A list of symbols containing the **name** of the contact, the **name** of the contact's **parent** (and so on), and the name of the **contact-display** to which the contact belongs. The complete resource name is one of the two items used as a key by a CLUE application in order to access a contact resource value in a resource database.

connection The IPC path between the server and client program. A client program typically has one connection to the server over which requests and events are sent.

constraint resources Resources defined for each child belonging to a member of a composite class. Constraint resources are typically used to control the parent's geometry management policy. Also see class resources.

contact The basic CLUE object for programming a user interface.

contact-display The CLUE object type that represents a connection to an X server and that supports an event loop for application input.

contact initialization The process of collecting initial values for all contact attributes. No server resources (windows and so on) are actually allocated until contact realization.

contact realization The process of allocating contact resources. This process completes contact creation.

containment A window contains the pointer if the window is viewable and the hot spot of the cursor is within a visible region of the window or a visible region of one of its inferiors. The border of the window is included as part of the window for containment. The pointer is in a window if the window contains the pointer but no inferior contains the pointer.

content The single child of a shell. The basic geometry management policy implemented by the **shell** class constrains a shell and its content to have the same width and height; size changes to one are automatically applied to the other.

coordinate system The coordinate system has x horizontal and y vertical, with the origin [0, 0] at the upper left. Coordinates are discrete and are in terms of pixels. Each window and pixmap has its own coordinate system. For a window, the origin is at the inside upper left, inside the border.

cursor The visible shape of the pointer on a screen. It consists of a hot-spot, a source bitmap, a shape bitmap, and a pair of colors. The cursor defined for a window controls the visible appearance when the pointer is in that window.

---

d

depth The depth of a window or pixmap is number of bits per pixel it has. The depth of a graphics context is the depth of the drawables it can be used in conjunction with for graphics output.

descendant If W is an inferior of A, then W is a *descendant* of A.

device Keyboards, mice, tablets, track-balls, button boxes, and so forth, are all collectively known as input *devices*. The core protocol only deals with two devices: the keyboard and the pointer.

direct color A class of colormap in which a pixel value is decomposed into three separate subfields for indexing. One subfield indexes an array to produce red intensity values, the second subfield indexes a second array to produce blue intensity values, and the third subfield indexes a third array to produce green intensity values. The RGB values can be changed dynamically.

dispatching an event The process of finding the appropriate contact and its actions.

double-click A sequence of two clicks of the same button in rapid succession.

drawable Both windows and pixmaps can be used as sources and destinations in graphics operations. These are collectively known as *drawables*. However, an **:input-only** window cannot be used as a source or destination in a graphics operation.

---

e

event Clients receive information asynchronously via *events*. These events can be either asynchronously generated from devices, or generated as side effects of client requests. Events are grouped into types; events are never sent to a client by the server unless the client has specifically asked to be informed of that type of event, but clients can force events to be sent to other clients. Events are typically reported relative to a window.

event compression Ignoring (or compressing) certain redundant input events. Compression of redundant events is controlled by the class slots **compress-exposures** and **compress-motion**, which are shared by all instances of a contact class.

event loop The fundamental application control structure: wait for an event, figure out how to handle it, process the event, then go back and wait for the next one. In CLUE, the event loop is implemented using the **process-next-event** function.

event mask Events are requested relative to a window. The set of event types a client requests relative to a window are described using an *event mask*.

event propagation Device-related events *propagate* from the source window to ancestor windows until some client has expressed interest in handling that type of event, or until the event is discarded explicitly.

event specification A notation for describing a certain sort of event. CLUE input processing consists of matching an event with an event specification found in a contact's **event-translations** slot and then calling actions associated with the matching event specification.

event synchronization Certain race conditions are possible when demultiplexing device events to clients (in particular deciding where pointer and keyboard events should be sent when in the middle of window management operations). The event synchronization mechanism allows synchronous processing of device events.

event source The smallest window containing the pointer is the *source* of a device related event.

event translation The process of determining which contact action functions will be executed. An event translation is a list found in a contact's **event-translations** slot associating an event specification with one or more action names. Also see class event translations.

exposure event Servers do not guarantee to preserve the contents of windows when windows are obscured or reconfigured. *Exposure* events are sent to clients to inform them when contents of regions of windows have been lost.

extension Named *extensions* to the core protocol can be defined to extend the system. Extension to output requests, resources, and event types are all possible, and expected.

---

f

focus window Another term for the input focus.

font A matrix of glyphs (typically characters). The protocol does no translation or interpretation of character sets. The client simply indicates values used to index the glyph array. A font contains additional metric information to determine inter-glyph and inter-line spacing.

---

g

geometry management The process whereby a composite controls the geometrical properties of its child contacts; the composite is referred to as the geometry manager.

glyph An image, typically of a character, in a font.

grab — Keyboard keys, the keyboard, pointer buttons, the pointer, and the server can be *grabbed* for exclusive use by a client. In general, these facilities are not intended to be used by normal applications but are intended for various input and window managers to implement various styles of user interfaces.

gcontext — Shorthand for graphics context.

graphics context — Various information for graphics output is stored in a *graphics context* (or gcontext), such as foreground pixel, background pixel, line width, clipping region, and so forth. A graphics context can only be used with drawables that have the same root and the same depth as the graphics context.

gray scale — A degenerate case of pseudo color, in which the red, green, and blue values in any given colormap entry are equal, thus producing shades of gray. The gray values can be changed dynamically.

---

h

hot spot — A cursor has an associated *hot spot* that defines a point in the cursor that corresponds to the coordinates reported for the pointer.

---

i

identifier — Each resource has an *identifier*, a unique value associated with it that clients use to name the resource. An identifier can be used over any connection to name the resource.

inferiors — All of the subwindows nested below a window: the children, the children's children, and so on.

initialization — See contact initialization.

input event — See event.

input focus — Normally a window defining the scope for processing of keyboard input. If a generated keyboard event would normally be reported to this window or one of its inferiors, the event is reported normally; otherwise, the event is reported with respect to the focus window. The input focus also can be set such that all keyboard events are discarded and that the focus window is dynamically taken to be the root window of whatever screen the pointer is on at each keyboard event.

input-only window — A window that cannot be used for graphics requests. *input-only* windows are invisible, and can be used to control such things as cursors, input event generation, and grabbing. *input-only* windows cannot have *input/output* windows as inferiors.

input/output window — The normal kind of opaque window, used for both input and output. Input/output windows can have both *input/output* and input-only windows as inferiors.

insensitivity — See sensitivity.

interactive-stream — A contact subclass designed to integrate CLUE with the conventional stream-based I/O of Common Lisp.

k

key grabbing
Keys on the keyboard can be passively *grabbed* by a client. When the key is pressed, the keyboard is then actively grabbed by the client.

keyboard grabbing
A client can actively *grab* control of the keyboard, and key events will be sent to that client rather than the client to which the events would normally have been sent.

keysym
An encoding of a symbol on a keycap on a keyboard.

m

managed
A contact under geometry management control.

mapped
A window is said to be *mapped* if a map call has been performed on it. Unmapped windows and their inferiors are never viewable or visible.

modifier keys
SHIFT, CONTROL, META, SUPER, HYPER, ALT, Compose, Apple, CAPS LOCK, Shift Lock, and similar keys are called *modifier keys*.

monochrome
A special case of static gray, in which there are only two colormap entries.

o

obscure
A window is *obscured* if some other window obscures it. For example, window A obscures window B if:

- Both windows are viewable **:input-output** windows
- Window A is higher in the global stacking order than window B
- The rectangle defined by the outside edges of window A intersects the rectangle defined by the outside edges of window B

Notice that window borders are included in the calculation, and that a window can be obscured and yet still have visible regions. See occlude (there is a fine distinction between obscure and occlude).

occlude
A window is *occluded* if some other window occludes it. For example, window A occludes window B if:

- Both windows are mapped
- Window A is higher in the global stacking order than window B
- The rectangle defined by the outside edges of window A intersects the rectangle defined by the outside edges of window B

Notice that window borders are included in the calculation. See obscure (there is a fine distinction between occlude and obscure).

override-shell A subclass of **shell** used to override the window manager. This subclass contains pop-up menus and other temporary objects that the user can never resize and so on.

p

padding Some padding bytes are inserted in the data stream to maintain alignment of the protocol requests on natural boundaries. This increases ease of portability to some machine architectures.

parent window If C is a child of P, then P is the *parent* of C.

passive grab Grabbing a key or button is a *passive grab*. The grab activates when the key or button is actually pressed.

pixel value An *n*-bit value, where *n* is the number of bit planes used in (that is, the depth of) a particular window or pixmap. For a window, a pixel value indexes a colormap to derive an actual color to be displayed.

pixmap A three dimensional array of bits. A pixmap is normally thought of as a two dimensional array of pixels, where each pixel can be a value from 0 to $(2^n) - 1$, where *n* is the depth (z axis) of the pixmap. A pixmap can also be thought of as a stack of *n* bitmaps.

plane When a pixmap or window is thought of as a stack of bitmaps, each bitmap is called a *plane* or *bit plane*.

plane mask Graphics operations can be restricted to only affect a subset of bit planes of a destination. A *plane mask* is a bit mask describing which planes are to be modified, and it is stored in a graphics context.

pointer The pointing device attached to the cursor and tracked on the screens.

pointer grabbing A client can actively *grab* control of the pointer, and button and motion events will be sent to that client rather than the client to which the events would normally have been sent.

pointing device Typically a mouse or tablet, or some other device with effective dimensional motion. There is only one visible cursor defined by the core protocol, and it tracks whatever pointing device is attached as the pointer.

pop-up One of the uses of a top-level shell (for example, a menu that pops up when a command button contact is activated). Setting the **state** of a shell to **:mapped** is sometimes referred to as *mapping* or *popping up* the shell. Setting the **state** of a shell to **:withdrawn** or **:iconic** is sometimes referred to as *unmapping* or *popping down* the shell.

property Windows can have associated *properties*, consisting of a name, a type, a data format, and some data. The protocol places no interpretation on properties; they are intended as a general-purpose naming mechanism for clients. For example, clients might share information such as resize hints, program names, and icon formats with a window manager via properties.

property list The list of properties that have been defined for a window.

pseudo color A class of colormap in which a pixel value indexes the colormap to produce independent red, green, and blue values. That is, the colormap is viewed as an array of triples (RGB values). The RGB values can be changed dynamically.

r

realization	See contact realization.
redirecting control	Window managers (or client programs) may choose to enforce window layout policy in various ways. When a client attempts to change the size or position of a window, the operation can be *redirected* to a specified client, rather than the operation actually being performed.
reply	Information requested by a client program is sent back to the client with a *reply.* Both events and replies are multiplexed on the same connection. Most requests do not generate replies. However, some requests generate multiple replies.
representation type	The type of representation of a resource value. For example, a color value might be represented either as a namestring ("red"), a pixel value, an RGB triplet, an HSV triplet, and so on.
request	A command to the server is called a *request*. It is a single block of data sent over a connection.
resource	A value of the user interface that can be changed by the user in a resource database via CLX functions **add-resource**, **get-resource**, and so forth. See server resource.
resource class, complete	See complete resource class.
resource database	Conceptually, a set of resource name/value pairs (or resource bindings). CLX defines functions for storing and retrieving interface resources from a resource database.
resource name, complete	See complete resource name.
RGB values	*Red*, *green*, and *blue* intensity values used to define color. These values are always represented as 16-bit unsigned numbers, with zero being the minimum intensity and 65535 being the maximum intensity. The values are scaled by the server to match the display hardware.
root	A special composite contact used to represent an entire display screen.
root window	Each screen has a *root window* covering it. It cannot be reconfigured or unmapped, but otherwise acts as a full-fledged window. A root window has no parent.

s

save set	The *save set* of a client is a list of other client's windows that, if they are inferiors of one of the client's windows at connection close, should not be destroyed and that should be remapped if it is unmapped. Save sets are typically used by window managers to avoid lost windows if the manager should terminate abnormally.
scanline	A list of pixel or bit values viewed as a horizontal row (all values having the same y coordinate) of an image, with the values ordered by increasing x coordinate.
scanline order	An image represented in *scanline order* contains scanlines ordered by increasing y coordinate.

screen A server can provide several independent *screens*, which typically have physically independent monitors. This would be the expected configuration when there is only a single keyboard and pointer shared among the screens.

selection A *selection* can be thought of as an indirect property with dynamic type. That is, rather than having the property stored in the server, it is maintained by some client (the *owner*). A selection is global in nature, being thought of as belonging to the user (but maintained by clients), rather than being private to a particular window subhierarchy or a particular set of clients. When a client asks for the contents of a selection, it specifies a selection *target type*. This target type can be used to control the transmitted representation of the contents.

For example, if the selection is "the last thing the user clicked on" and that is currently an image, then the target type might specify whether the contents of the image should be sent in XY Format or Z Format. The target type can also be used to control the class of contents transmitted; that is, asking for the looks (fonts, line spacing, indentation, and so forth) of a paragraph selection, rather than the text of the paragraph. The target type can also be used for other purposes; the semantics is not constrained by the protocol.

sensitivity A condition in which a user interface component of an application will accept input. Conversely, when a contact is insensitive, events of particular types are not dispatched to the contact and are ignored.

server The *server* provides the basic windowing mechanism. It handles IPC connections from clients, demultiplexes graphics requests onto the screens, and multiplexes input back to the appropriate clients.

server grabbing The server can be *grabbed* by a single client for exclusive use. This prevents processing of any requests from other client connections until the grab is complete. This is typically only a transient state for such things as rubber-banding and pop-up menus, or to execute requests indivisibly.

server resource Windows, pixmaps, cursors, fonts, gcontexts, and colormaps are known as resources. They all have unique identifiers associated with them for naming purposes. The lifetime of a resource is bounded by the lifetime of the connection over which the resource was created. See resource.

shell A composite that handles the duties required by standard conventions for top-level X windows.

sibling Children of the same parent window are known as *sibling* windows.

static color A degenerate case of pseudo color in which the RGB values are predefined and read-only.

static gray A degenerate case of gray scale in which the gray values are predefined and read-only. The values are typically (near-)linear increasing ramps.

stacking order Sibling windows can *stack* on top of each other. Windows above both obscure and occlude lower windows. This is similar to paper on a desk. The relationship between sibling windows is known as the *stacking order.*

state A slot of **contact** that controls the visual effect of the contact.

stipple A bitmap that is used to tile a region to serve as an additional clip mask for a fill operation with the foreground color.

t

tile A pixmap can be replicated in two dimensions to *tile* a region. The pixmap itself is also known as a tile.

timer A CLUE object that provides support for animation and other types of time-sensitive user interfaces. A timer causes **:timer** events to be dispatched to a specific contact for processing.

timestamp A time value, expressed in milliseconds, typically since the last server reset. Timestamp values wrap around (after about 49.7 days). The server, given its current time is represented by timestamp T, always interprets timestamps from clients by treating half of the timestamp space as being earlier in time than T and half of the timestamp space as being later in time than T. One timestamp value (named CurrentTime) is never generated by the server; this value is reserved for use in requests to represent the current server time.

top-level contact A contact whose parent is a root. A top-level contact is usually a composite at the top of a hierarchy of other contacts created by an application program.

top-level-session A subclass of **shell** that is used to communicate with a session manager.

top-level-shell A subclass of **shell** that provides full window manager interaction.

transient-shell A subclass of **shell** that a window manager typically will unmap when its owner becomes unmapped or iconified and will not allow to be individually iconified.

true color A degenerate case of direct color in which the subfields in the pixel value directly encode the corresponding RGB values. That is, the colormap has predefined read-only RGB values. The values are typically (near-)linear increasing ramps.

type An arbitrary atom used to identify the interpretation of property data. Types are completely uninterpreted by the server; they are solely for the benefit of clients.

u

unmanaged A contact that is not under geometry management control.

user interface A set of abstract interface objects used to control the dialog between an application and its human user.

v

viewable A window is *viewable* if it and all of its ancestors are mapped. This does not imply that any portion of the window is actually visible. Graphics requests can be performed on a window when it is not viewable, but output will not be retained unless the server is maintaining backing store.

visible A region of a window is *visible* if someone looking at the screen can actually see it; that is, the window is viewable and the region is not occluded by any other window.

W

window gravity When windows are resized, subwindows can be repositioned automatically relative to some position in the window. This attraction of a subwindow to some part of its parent is known as *window gravity*.

window manager Manipulation of windows on the screen, and much of the user interface (policy) is typically provided by a *window manager* client.

window manager shell A subclass of **shell** called **wm-shell** that interacts with the window manager.

X

XY Format The data for a pixmap is said to be in *XY Format* if it is organized as a set of bitmaps representing individual bit planes, with the planes appearing from most to least significant in bit order.

Z

Z Format The data for a pixmap is said to be in *Z Format* if it is organized as a set of pixel values in scanline order.

General

D

E

F

G

T

U

V

W

X

Conditions

A

C

D

F

G

I

L

M

N

P

R

S

U

V

W

X

Functions

A

B

C

D

E

F

G

H

I

K

L

M

O

P

Q

R

S

T

U

W

Types

A

B

C

D

E

F

G

I

K

M

P