X-Icon: An Icon Window Interface [1]
Version 2

Clinton L. Jeffery

TR 92-26

Abstract

This document describes the calling interface and usage conventions of X-Icon, a window system interface for the Icon programming language that supports high-level graphical user interface programming. It presently runs on UNIX and VMS systems under Version 11 of the X Window System, and on OS/2 2.0 under Presentation Manager.

November 7, 1992

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1] This work was supported in part by the National Science Foundation under Grant CCR-8713690 and a grant from the AT&T Research Foundation.

Introduction

This document is intended for programmers writing window programs using X-Icon, an interface to graphical user interface facilities for the Icon programming language. This document corresponds to Version 8.8 of Icon. The document consists of two parts, a user's guide followed by a reference section. The user's guide describes the interface and presents several examples of its use; the reference section includes a complete description of all functions in the interface. Lest there be any confusion later on, the term "Icon" in this document denotes the Icon programming language [Gris90]. We use lowercase letters to denote those little pictures on the computer screen, e.g. "icon".

X-Icon adds an interface to the raster graphics, text fonts, colors, and mouse input facilities provided by graphic interfaces such as the X Window System[1] [Sche86]. Because different hardware and different window system software have different capabilities, there is a trade-off between simplicity, portability, and full access to the underlying machine. Most window system interfaces are complex mazes that require vast amounts of training and experience in order to program effectively. Unlike other languages' window interfaces, X-Icon leans towards simplicity and ease of programming rather than full low-level access. Nevertheless, a basic knowledge of window system concepts will be useful in understanding what follows.

The Programming Interface

The X-Icon interface is loosely based on Xlib [Get88, Nye 88], the low-level C interface for X Window programming. With minor exceptions, this same interface is implemented for OS/2, even though the underlying window system calls are different.

Like Xlib, X-Icon provides a windowing mechanism without enforcing a particular policy, user interface, or look-and-feel. Although based on Xlib, X-Icon provides a higher-level abstraction more consonant with the rest of the Icon language. The run-time system implements retained windows, automatically redrawing obscured portions of windows when they become visible; events other than keystrokes and mouse events are similarly handled automatically. Higher-level toolkits and libraries implementing advanced user interface features can be written in Icon.

Most window-system interfaces are *event driven*, meaning that they present a paradigm in which an event-reading loop is the primary control mechanism driving the application. For example, if an application must be prepared to redraw the contents of its windows at all times, it cannot compute for long periods without checking for window events.

Although this event-driven paradigm is central in the underlying implementation, it is *optional* in the X-Icon programming model. Since X-Icon's windows handle many events automatically and "take care of themselves", applications follow the event-driven paradigm only when it is needed and appropriate. Simple window applications can be written using very ordinary-looking code.

Icon's dynamic typing and polymorphic approach to control structures and operations greatly reduce the burden on the user interface programmer. Icon's standard file I/O routines all work on windows, and greatly simplify text-oriented applications. Arguments to graphics routines are simplified because the display and selected graphics context are implied by the window argument. The event-reading function produces

[1]The X Window System is a trademark of the Massachusetts Institute of Technology.

different types of values for different types of events: keystrokes are strings, and mouse events are integers. Icon's polymorphic case expression allows keyboard and mouse events to be handled conveniently in a unified fashion.

The above properties combined with the extensive use of default values make simple window applications extremely easy to program, while providing flexibility where it is needed in more sophisticated applications.

The File Device Model

Windows are a special file data type opened with mode x; thus, a simple X-Icon program might look like this:

```
procedure main()
    w := open("helloTool", "x")
    write(w, "hello, world")
    # do processing ...
    close(w)
end
```

A window appears on the screen as a rectangular space for text and/or graphics. Windows (files opened with mode x) are open for both reading and writing. They support the usual file operations with the exceptions of seek() and where(); given window arguments, these functions simply fail. The type() of a window is "window"; the image is "window(*windowname*)". Like other files, windows close automatically when the program terminates, so the call to close() in the above example is optional.

At this point enough has been covered to write a first useful X-Icon program: *xprompt*. *Xprompt* is a utility for shell programmers who occasionally need to retrieve a filename or other simple input from the user when the shell script is run. The program pops up a window on the user's screen asking a question, reads the user's answer, and writes it out on its standard output where it can be accessed by the shell script by means of the backquotes. For example,

```
biff `xprompt "Shall I turn on biff for you? "`
```

Xprompt uses the file model and Icon's built-in file operations to do all the work:

```
procedure main(args)
    w := open("xprompt", "x") | stop("can't open window")
    every writes(w, !args, " ")
    write(read(w))
end
```

This example is fully operational and is not unlike a standard Icon program that might perform the same task. In an X environment, there may be dozens of windows with programs competing for the user's attention, and this is where *xprompt* is more useful than an ordinary Icon program. Popping up a prompting window on the user's screen attracts quite a bit more attention than a standard Icon program — especially if the shell script that is executing is running a window that has been obscured behind another window or reduced to iconic size to save screen space. This version of *xprompt* is visually unappealing because the default window size is inappropriate, and the default font used is too small. A nicer looking version is given below.

Text Coordinates

Windows are a variant of the file data type, but more specifically they are modeled after the standard computer terminal text screen and support features such as scrolling and cursor positioning. These features employ a coordinate system that describes screen positions in terms of the row and column within the text that is in view in the window. The rows are counted from the top of the screen to the bottom and the columns are counted from the left of the screen to the right. Row and column coordinates are *one-based*; that is, the very upper left corner of the screen is text position (1,1). The function XGotoRC(w, r, c) moves the text cursor for w to row r, column c.

Window Attributes

A window's state consists of several *attributes*. Each attribute has an associated *value*. Some values are defined by external forces in the window system (such as the user or window manager), while others are under program control. In the absence of program-supplied values, attributes are assigned default values.

Icon's standard function open() has been extended to allow any number of string arguments after the file name and mode arguments. The format of these strings is more precisely described in the next section. These arguments specify initial values of attributes when the window is created. For example, to say hello in italics on a window with blue background one can write:

```
procedure main()
    w := open("helloTool", "x", "font=italics", "bg=blue")
    write(w, "hello, world")
    # processing ...
end
```

In order for this code to run as-is, there must be a window system font named italics.

To use a more concrete example, window attributes allow *xprompt* to use a larger, more readable font, and display itself in a window exactly one line high:

```
procedure main(args)
    w := open("xprompt", "x", "font=12x24", "lines=1") | stop("can't xprompt")
    every writes(w, !args, " ")
    write(read(w))
end
```

Xprompt might attract even more attention to itself on a color monitor by utilizing bright foreground and background colors instead of the default black-on-white.

After a window is created, its attributes may be inspected and set using the Icon function XAttrib(w, s1, s2, ...). XAttrib() either gets or sets window attributes according to the string arguments. Certain attributes can only be read by XAttrib() and not set.

Window Attribute-Values

Attributes are read and written as strings of the form "*attr*[=*val*]", e.g.

```
w := open("Hello", "x", "lines=24", "columns=80")
write(w, "Hello, world")
XAttrib(w, "fg=red", "bg=green", "font=italics", "row=12")
write(w, "Goodbye ...")
```

Arguments to XAttrib() and open() that include an equals sign are *assignments*; the attribute is set to the given value if possible, but XAttrib() fails otherwise. open() only contains attribute assignments of this form. XAttrib() generates a string result for each of its arguments; in the case of assignment, the result is the same "*attr*=*val*" form the arguments take. Attributes are also frequently set by the user's manipulation of the window; for instance, cursor or mouse location or window size.

String arguments to XAttrib() that consist only of an attribute are *queries*. When multiple queries are made via a single call to XAttrib(), each answer is generated in turn as the attribute suffixed by the appropriate =val value. Results from multiple queries are generated in the order in which the arguments were passed to XAttrib(). Thus, XAttrib(w, "lines", "columns") generates two results, for example, "lines=25" followed by "columns=80".

In the common case in which a single attribute query is made, the attribute name and equals sign are omitted from the return string since the attribute is not ambiguous. In the example above, XAttrib(w, "lines") returns the string "25".

Xm: a File Browser

Using attributes leads to applications that are still similar to ordinary text applications, but begin to exhibit special properties. *Xm* is a trivial X-Icon version of the famous pager utility named *more*. *Xm* displays a text file (its first argument) in a window, one screenful at a time, and allows the user to scroll forward and backward through the document. This simple version of *xm* is less flexible than *more* in most ways (it is written after all in less than thirty lines of code), but it gains certain flexibility for free from its window system orientation: The window can be resized at any time, and *xm* takes advantage of the new window size after the next keystroke. The complete text of *xm* is presented in Appendix A.

Xm begins with the lines

```
procedure main(argv)
   if *argv = 0 then stop("usage: xm file")
```

These two lines simply start the main procedure and issue a message if the program has been invoked with no filename. A more robust version of *xm* would handle this case in the proper UNIX fashion by reading from its standard input.

After its (scant) argument checking, *xm* opens the file to be read, followed by a window to display it in.

```
f := open(argv[1], "r") | stop("can't open ", argv[1])
w := open(argv[1], "x") | stop("no window")
```

If either of these values cannot be obtained, *xm* gives up and prints an error message.

With an open file and window in hand, *xm* is ready to read in the file. It does so in brute-force fashion, reading in all the lines at once and placing them in a list.

```
L := []
every put(L, !f)
close(f)
```

A more intelligent approach would be to read the file gradually as the user requests pages. This approach makes no difference for short files but is superior for very large files.

At this point, *xm* has done all of its preparatory work, and is ready to display the first page of the file on the screen. After displaying the page, it waits for the user to press a keystroke: a space bar indicates the next page should be displayed, the "b" key backs up and displays the preceding page, and the "q" key quits the application. The overall structure looks like:

```
repeat {
   # display the current page of text in the file
   # read and execute a one-keystroke command
   }
```

Xm writes out pages of text to the screen by indexing the list of lines L. The index of the first line that is displayed on the screen is remembered in variable base, and paging operations are performed as arithmetic on this variable.

```
XClearArea(w)
XGotoRC(w, 1, 1)
every i:= 0 to XAttrib(w, "lines") - 1 do {
   if i + base < *L then writes(w, L[i + base + 1])
   write(w)
   }
```

UNIX *more* writes a nice status line at the bottom of the screen, indicating where the current page is within the file as a percentage of the total file size. Writing this line out in reverse video is a matter of calling XAttrib() before and afterwards. Computing the percentage is done based on the last text line on the screen (base + XAttrib(w, "lines") - 1) rather than the first.

```
XAttrib(w, "reverse=on")
writes(w, "--More--(",
         ((100 > (base + XAttrib(w,"lines") - 1) * 100 / *L) | 100),
         "%)")
XAttrib(w, "reverse=off")
```

Keystrokes are read from the user using Icon's regular built-in function reads().

```
case reads(w, 1) of {
    "q": break
    " ": base := (*L > (base + XAttrib(w, "lines") – 1) | fail)
    "b": base := (0 < (base – XAttrib(w, "lines") + 1) | 0)
    }
```

Xm demonstrates that X-Icon demands little or no window system expertise of the Icon programmer. Ordinary text applications can be ported to X with very few changes by adding a window argument to calls to functions such as read() and write(). After a program has been ported, it is simple to enhance it with attributes such as colors and fonts. Other more subtle output attributes are discussed later in the section on graphics contexts.

A Bitmapped-Graphics Device Model

Built-in functions corresponding directly to a subset of the Xlib interface provide X-Icon programmers with convenient access to bitmapped graphics capabilities. These facilities constitute a second programming model for windows, but there are no programming "modes" and code that uses graphics may be freely intermixed with code that performs text operations. There are many graphics functions, and they are detailed in the reference section of this document. The reader should consult Xlib documentation (see, for example, [Nye88]) for the precise semantics of many of these calls.

Among graphics functions, one major addition to Xlib's functionality is the XDrawCurve() function. This function draws smooth curves through specified points.

&window: The Default Window Argument

Just as Icon has keywords that supply default files for text input and output, X-Icon has a keyword, &window, that supplies a default window for graphic input and output. &window is like &subject in that it is a variable; it starts out with a value of &null and can be assigned to using Icon's regular assignment operator. Only window values (and the null value) may be assigned to &window. &window serves as a default argument to most X-Icon functions and is used implicitly by various operations. Exceptions are noted in the reference section.

In the previous program example, if *xm* used &window instead of the variable w, the argument could be omitted from the calls to XClearArea() and XAttrib(). The window argument still has to be supplied for calls to normal file functions such as write() and writes() since these functions default to &output instead of &window. Window argument defaulting is not very important in the preceding example, but it is quite important for more graphics-oriented programs where it can both shorten the code and make it faster.

Graphics Coordinates

A coordinate system was introduced earlier for text-handling; that coordinate system was defined in terms of rows and columns of text, and was used primarily to talk about positioning text on the screen in a manner similar to a standard computer terminal text-display. None of the graphics functions affect the text cursor, and text and graphics can be mixed freely.

The graphics functions use an integer coordinate system that counts individual *pixels* (picture elements, or dots). Like the text coordinate system, graphics coordinates start in the upper left corner of the screen. From that corner the positive x direction lies to the right and the positive y direction moves down. Unlike the text coordinate system, the graphics coordinate axes are *zero-based*, which is to say that the very top leftmost pixel is (0,0).

Converting Between Graphics and Text Coordinates

X-Icon uses four integer valued keywords to generically refer to coordinates: &x and &y refer to pixel coordinates and &row and &col refer to text coordinates. &x and &col are coupled since they both refer to horizontal position in their respective coordinate systems; similarly &y and &row are coupled.

Generally, these values will be assigned by X-Icon and read by the Icon program as a result of various operations described later. Sometimes, however, it is useful to convert between text and graphics coordinates in some window. Assigning to &x, &y, &row, or &col has the effect of assigning a converted value to the appropriate coupled keyword in the other coordinate system. The converted values are computed in &window using its current font. For example, the following code computes the graphic coordinates corresponding to row 10, column 10 in the current font, and prints a pair of values such as 54,128.

```
&row := &col := 10
write(&x, ",", &y)
```

Events

All user input actions including keystrokes and mouse clicks are termed *events*. In X-Icon many events are handled by the run-time system without intervention of the programmer. These include window redrawing and resizing, etc. Other events are put on an *event queue* in the order they take place, for processing by the Icon program.

Regular keyboard events are communicated to the Icon program through any of the standard file input functions (for example, reads(w, 1)). When reading from a window using the standard input functions, only keyboard events are available; mouse and special key events that are processed during standard input on a window are dropped.

In addition to being closed by the function close() or by program termination, under many window systems a window may be *killed* by the user or window manager; when a window is killed in this manner, the Icon program executing it terminates with runtime error number 141 (“program terminated by window manager”).

Event Queue Manipulation

The event queue is an Icon list that stores events until the program processes them. When a user presses a key, clicks or drags a mouse, or resizes a window, three values are placed on the event queue: the event itself, followed by two integers containing associated event information. Events are removed from the queue by built-in input functions including read() and reads().

In addition to the "cooked" input functions read() and reads(), two "raw" input functions are defined for windows. XPending(w) produces the event queue, while XEvent(w) produces the next event for window w and removes it from the queue.

If no events are pending, the list returned by XPending() is empty. If events are pending, the number of elements on the event queue is normally *XPending(w) / 3. The list returned by XPending() is special in that it remains attached to the window, and additional events may be added to it at any time during program execution. In other respects it is an ordinary list and can be manipulated using Icon's list functions and operators. Details of the list format are presented in Appendix E.

Since the list returned by XPending() remains attached to the window and is used by subsequent calls to XEvent(), it can be used to achieve a variety of effects such as simulating key or mouse activity within the application. The ordinary list function push() is all that is required to insert events at the head of the queue (that is, so that they are the next events to be read). Inserting events at the tail of the queue is complicated by the fact that the window system can add events between calls to put() that the program might make. XPending(w,x_1,...,x_n) adds x_1 through x_n to the end of w's pending list in guaranteed consecutive order.

The X-Icon function XEvent(w) allows the Icon program to read the next keyboard or mouse event. Generally, keyboard events are returned as strings, while mouse events are returned as integers and are described more fully below. Special keys, such as function keys and arrow keys, are also returned as integers, described below.

If no events are pending, XEvent() waits for an available event. Once an event is available, XEvent() removes an element from the queue and produces it as a return value. In addition, XEvent() removes the next two elements and assigns to the keywords &x and &y the x and y pixel coordinates of the mouse at the time of the event. XEvent() also assigns the mouse location to keywords &row and &col in text row and column coordinates; these are provided solely for the convenience of text-oriented applications as they could be extracted from &x and &y by indirect means. The values of &x, &y, &row, and &col remain available until a subsequent call to XEvent() again assigns to them.

XEvent() sets the keyword &interval to the number of milliseconds that have elapsed since the previous event (or to zero for the first event). The keywords &control, &shift, and &meta are set by XEvent() to return the null value if the respective modifier keys were pressed at the time of the event; otherwise they fail. For resize events, &interval is set to zero and modifier keywords fail.

Keyword side effects associated with event processing on windows are summarized in the following table:

Keyword	Description
&x	mouse location, horizontal
&y	mouse location, vertical
&row	mouse location, text row
&col	mouse location, text column
&interval	time since last event, milliseconds
&control	succeeds if control key pressed
&shift	succeeds if shift key pressed
&meta	succeeds if meta key pressed

Frequently when several windows are open, the program needs to await user activity on any of the windows and handle it appropriately. Although this could be implemented by repeatedly examining each window's pending list until a nonempty list is found, such a busy-waiting solution is wasteful of CPU time. The function XActive() waits for window activity, relinquishing the CPU until an event is pending on one of the open windows, and then returns a window with a pending event. XActive() cycles through the open windows on repeated calls in a way that avoids window *starvation*. A window is said to starve if its pending events are never serviced.

Mouse Events

Mouse events are returned from XEvent() as integers indicating the type of event, the button involved, etc. Keywords allow the programmer to treat mouse events symbolically. The event keywords are:

Keyword	X Event
&lpress	mouse press left
&mpress	mouse press middle
&rpress	mouse press right
&lrelease	mouse release left
&mrelease	mouse release middle
&rrelease	mouse release right
&ldrag	mouse drag left
&mdrag	mouse drag middle
&rdrag	mouse drag right
&resize	window was resized

The following program uses mouse events to draw a box that follows the mouse pointer around the screen when a mouse button is pressed. The attribute drawop=reverse allows drawing operations to serve as their own inverse and is described later. Function XFillRectangle() draws a filled rectangle on the window and is described in the reference section. Each time through the loop the program erases the box at its old location and redraws it at its new location; the first time through the loop there is no box to erase so the first call to

XFillRectangle is forced to fail by means of Icon's \ operator.

```
procedure main()
    &window := open("hello", "x", "drawop=reverse")
    repeat if XEvent() === (&ldrag | &mdrag | &rdrag) then {
        XFillRectangle(\x, \y, 10, 10)          # erase box at old position
        XFillRectangle(x := &x, y := &y, 10, 10)   # draw box at new position
        }
end
```

The Icon program can inspect the rest of the window's state using XAttrib(). Between the time the mouse event occurs and the time it is produced by XEvent(), the mouse may have moved. In order to get the current mouse location, use XQueryPointer() (see below).

When more than one button is depressed as the drag occurs, drag events are reported on the most recently pressed button. This behavior is invariant over all combinations of presses and releases of all three buttons.

Resize events are not mouse events, but they are reported in the same format. In addition to the event code, &x, &y, &col and &row are assigned integers that indicate the window's new width and height in pixels and in text columns and rows, respectively.

Bme: a Bitmap Editor

A simple bitmap editor, *bme*, demonstrates event processing including mouse events. It displays both a small and a "magnified" display of the bitmap being edited, allows the user to set individual bits, and allows the user to save the bitmap. *Bme* consists of three procedures. It employs several graphics functions; the reader is encouraged to consult the reference section for descriptions of those functions. The complete text of *bme* is presented in Appendix A.

Bme starts by declaring and initializing several variables. w1 and w2 are the magnified and regular-sized windows, respectively. WIDTH and HEIGHT store the bitmap's dimensions. WIDTH and HEIGHT default to 32.

```
procedure main(argv)
    WIDTH := HEIGHT := 32
```

The bitmap's width and height can be specified on the command line with a −geo option, e.g. bme −geo 16x64. Geometry arguments are popped off the argument list if they are present.

```
    if argv[1] == "-geo" then {
        pop(argv)           # pop "-geo"
        argv[1] ? {
            WIDTH := integer(tab(many(&digits))) | stop("bad geo syntax")
            ="x" | stop("bad geo syntax")
            HEIGHT := integer(tab(0)) | stop("bad geo syntax")
            pop(argv)          # pop arg, e.g. 16x64
```

```
        }
    }
```

Following the geometry arguments, *Bme* proceeds to check for a supplied file argument specifying the bitmap to edit. If one is found, it is read into the regular scale window w, and then the magnified scale window is constructed.

In order to construct the magnified scale window, each pixel is copied (repeatedly) into the corresponding pixels in the expanded version of the image. An alternative would be to use the function XPixel() to read the contents of the regular scale window.

```
# Construct magnified copy of bitmap
every i := 0 to HEIGHT – 1 do {
    every j := 0 to WIDTH – 1 do {
        XCopyArea(w2, w1, j, i, 1, 1, j * 10, i * 10)
        XCopyArea(w2, w1, j, i, 1, 1, j * 10 + 1, i * 10)
        every k := 1 to 4 do {
            XCopyArea(w1, w1, j * 10, i * 10, 2, 1, j * 10 + k * 2, i * 10)
            }
        XCopyArea(w1, w1, j * 10, i * 10, 10, 1, j * 10, i * 10 + 1)
        every k := 1 to 4 do {
            XCopyArea(w1, w1, j * 10, i * 10, 10, 2, j * 10, i * 10 + k * 2)
            }
        }
    }
```

After the windows are loaded with their initial contents, if any, a grid is drawn on the magnified image to delineate each individual pixel's boundary. The user's mouse actions within these boxes turn them white or black.

The main event processing loop of *bme* is very simple: Each event is fetched with a call to XEvent() and immediately passed into a case expression. The keystroke "q" exits the program; the keystroke "s" saves the bitmap in a file by calling XWriteImage(), asking for a file name if one has not yet been supplied.

```
case e := XEvent(w1) of {
    "q": return
    "s": {
        /s := getfilename()
        XWriteImage(w2, s)
        }
```

Mouse events all result in the drawing of a black (for the left button) or white (for the right button) dot in both the magnified and regular scale bitmaps.

```
        &lpress | &ldrag: {
            dot(w1, w2, &x / 10, &y / 10, 1)
            }
```

Drawing of black and white dots is handled identically by procedure dot(), whose optional fifth argument specifies a black dot (if it is present) or a white dot (if it is absent). The dot is drawn using XFillRectangle() in the magnified window; in the regular scale window XDrawPoint() suffices.

```
procedure dot(w1, w2, x, y, black)
    if \black then {
        XFg(w1, "black")
        XFg(w2, "black")
        }
    else {
        XFg(w1, "white")
        XFg(w2, "white")
        }
    XFillRectangle(w1, x * 10, y * 10, 10, 10)
    XDrawPoint(w2, x, y)
    XFg(w1, "black")
    if /black then XDrawRectangle(w1, x * 10, y * 10, 10, 10)
end
```

Bme illustrates basic X-Icon event-handling: a single case expression that handles various keystrokes and mouse events as different cases makes the control structure simpler than in other languages' event processing.

Special Keys

The regular keys that X-Icon returns as one-letter strings correspond approximately to the lower 128 characters of the ASCII character set. These characters include the control keys and the escape character. Modern keyboards have many additional keys, such as function keys, arrow keys, "page down", etc. X-Icon produces integer events for these special keys; the integer values correspond to X window *keysyms*. A selection of the more common keysyms are given in Appendix D. The complete collection of keysym values are defined in the X include file <X11/keysymdef.h>.

Graphics Contexts

Some attributes are associated with the window itself, while others are associated with the *graphics context* used by operations that write to windows. Although this distinction is not necessary in simple applications such as *xm*, it becomes useful in more sophisticated applications that use multiple windows or draw many kinds of things in windows. A graphics context consists of the colors, patterns, line styles, and text fonts and sizes.

Although they are called graphics contexts, text mode operations use these attributes too: Text is written using the foreground and background colors and the font defined in the graphics context performing the operation. Table 2 in the reference section lists those attributes associated with a graphics context.

Binding Windows and Graphics Contexts Together

Graphics contexts can be shared among windows, and different graphics contexts can be used to write to the same window. An X-Icon window value is actually a *binding* of a window on the screen (a *system* window) with a particular graphics context. When `open(s,"x")` is called, it creates both a system window, and a context, and binds them together, producing the binding as its return value.

The built-in function `XBind()` is used to manipulate bindings. `XBind(`w_1, w_2`)` creates a new window value consisting of the window associated with w_1 bound to the graphics context associated with w_2. If w_2 is omitted, a new graphics context is allocated. The new context starts out with attributes identical to those of w_1's context.

If both window arguments are omitted, the binding produced has no associated system window on the screen; it is an invisible *pixmap* that can be used to manipulate images without showing them on-screen. Such images can then be copied onto visible windows with `XCopyArea()`. The default size of the pixmap is the same as the default window size.

Following any window arguments, `XBind()` accepts any number of string attributes to apply to the new window value, as in `open()` and `XAttrib()`.

After calling `XBind()`, two or more Icon window values can write to the same system window. The cursor location is associated with the window and not the graphics context, so writing to one window and then the other produces concatenated (rather than overlapping) output by default. Closing one of those window values removes the system window from the screen but does not close the logical window; at that point the remaining binding references an invisible pixmap. The logical window closes after the last binding associated with it closes.

Use of `XBind()` can significantly enhance performance for applications that otherwise require frequent graphics context manipulations (see Figure 1). It can also promote consistency in the look and feel of several related windows.

Coordinate Translation

A graphics context is a structure that chiefly consists of a set of graphics attributes that are used during drawing operations. In addition to these attributes, contexts have two attributes that perform output coordinate *translation*, `dx` and `dy`. `dx` and `dy` take integer values and default to zero. These integers are added into the coordinates of all *output* operations that use the context; input coordinates in `&x` and `&y` are *not* translated.

Attribute Types

Every attribute has an associated *attribute type* that defines the range of values that the attribute may take. Although all attribute values are encoded as strings, they represent very different window system features.

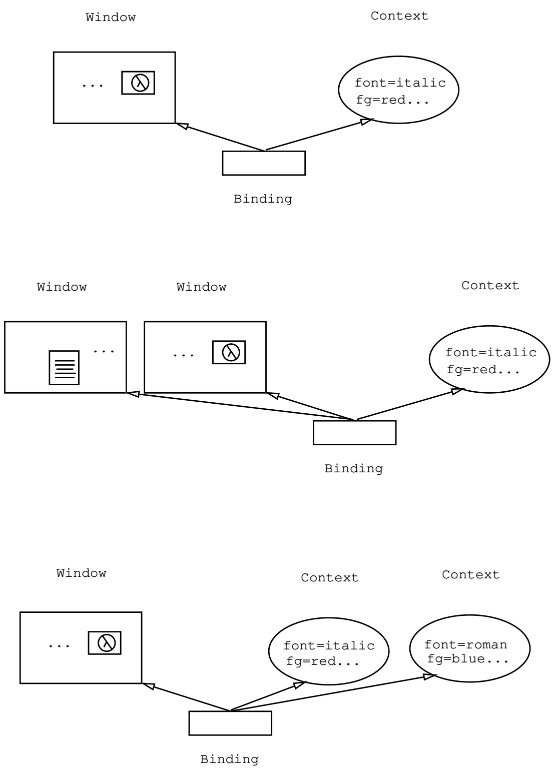

Figure 1: Some Uses of Bindings

Table 3 in the reference section lists the different attribute types and their values. This section provides additional details on some of the attribute types.

The attribute pointer refers to mouse pointer shapes that may be changed by the application during different operations. When the pointer attribute is set, the mouse pointer takes on an associated shape whenever the mouse is within the window in question. The available mouse pointers depend on the window system in use; see Appendices B and C for system-specific notes.

The attribute iconic takes a value indicating the window's *iconic state*. Windows with iconic state "root" are references to the window system's *root window* and cannot be moved or resized. Such windows typically are not used for normal drawing operations; the root window does not preserve its contents when obscured by other windows.

Text font and the foreground and background colors are specified using whatever strings are recognized by the window system software being utilized. See Appendices B and C for system-specific notes. In addition, the foreground and background may be specified by strings encoding the red, green, and blue components of the desired color. X-Icon accepts the hex formats "#*rgb*", "#*rrggbb*", "#*rrrgggbbb*", and "#*rrrrggggbbbb*", in which *r*, *g*, and *b* are 1 to 4 hex digits each, encoding the most significant bits of their respective components. Red, green, and blue may also be given in decimal format, separated by commas. The components are given on a scale from 0 to 65535 used internally by X, although color X servers typically offer far less precision. For example, "bg=32000,0,0" requests a dark red background; if the X server is incapable of such, it approximates it as closely as possible from the available colors. "fg=0,65000,0" requests a light green foreground.

The attribute pos refers to the position of the upper-left corner of the window on the screen. Screen position is specified by a string containing an x and a y coordinate separated by a comma, e.g. "pos=200,200".

Drawing Operations

At the individual pixel level, all text and graphic operations amount to a combination of some source bits with the bits that are already there. By default, drawing operations consist of the source bits overwriting whatever was present in the window beforehand. This behavior is not always what is desired.

Drawing operations are the sixteen possible logical combinations of the bits of the source (the bits to be drawn) and the destination (the bits already in the window). Drawing operations are specified by their string names assigned to the graphics context attribute drawop, which has a default value of "copy". The standard Xlib drawing operations are available using the names given in the table below. Operations other than "copy" are potentially nonportable or even undefined and should be used only with a clear understanding of the X color model.

In addition to the sixteen standard X drawing operations, there is one special drawop value that provides a portable higher-level reversible drawing operation. "drawop=reverse" changes pixels that are the foreground color to the background color, and vice-versa. The color it draws on pixels that are neither the foreground nor the background color is undefined. In any case, drawing a pixel a second time with "drawop=reverse" restores the pixel to its original color.

In Xlib terms, "drawop=reverse" draws using the bit-wise xor'ed value of the current foreground and background colors as its source bits. It combines these bits with the destination using an xor function. These two features allow it to work with all foreground and background colors on all display hardware.

"and"	"andInverted"	"andReverse"	"clear"	"copy"	"copyInverted"
"equiv"	"invert"	"nand"	"noop"	"nor"	"reverse"
"or"	"orInverted"	"orReverse"	"set"	"xor"	

Mutable Colors

The standard color allocation mechanism takes an *rgb* color value and allocates an entry for that color in a *color map*. Such values may be shared when more than one application needs the same color. X terms these color map entries as "shared" or "read-only".

Some color hardware devices are able to dynamically change the *rgb* color values for color map entries. Without updating all of display RAM, the colors of such entries can be changed almost instantaneously. X terms these as "mutable" or "read-write" color map entries. This mechanism can be used to produce a variety of special effects on those color systems that support it.

The function XNewColor(w) allocates a mutable color table entry on w's display if one is available, and fails otherwise. XNewColor(w, r, g, b) initializes that color table entry to a color given by r, g and b integer red, green and blue components, while XNewColor(w, s) initializes the color table entry to a color given by string color name as in XAttrib(). XNewColor() returns a small negative integer that may be used in calls to XFg(), XBg(), and XAttrib(). The mutable color may then be changed via the function XColor(w, n, x, g, b) where n is the small negative integer returned by XNewColor().

Window Icons

Each window is at any given instant in one of two possible *states*: it is either full-sized, or it is a miniature *iconic* window that often is suggestive of the contents of the full-sized window. Ordinarily window state changes are controlled by the user and are outside of program control; certain special mouse events on a window are sent not to the window but to a program called a *window manager* that iconifies or enlarges a window, moves it, resizes it, etc.

An X-Icon program need not be aware of the states of its windows, nor need it be concerned about what icons are used to represent them, since this is the job of the window manager (who in turn takes its instructions from the X user). Because the programmer often has a better idea of what an appropriate icon for a particular application would be, X-Icon includes a function, XIconImage(w, s), to specify an image to be used as a window's icon. s is a string file name of an image to use as the icon; the file may be either a standard X bitmap or an XPM pixmap [LeHo91].

Similarly, a function XIconic(w, s) is provided to allow the program to request that a window be in the state given by s, one of "icon" or "window". Another function, XMoveIcon(w, x, y) requests that window w's icon be moved to coordinates x, y. These functions are subject to the cooperation of the window manager in use; using these features in an Icon program limits its portability.

Etch: a Drawing Program

This section presents a view of the X-Icon interface via a more extended programming example. The full text of a drawing program, called *etch*, is presented in Appendix A. The Icon program library includes a more interesting version of *etch* that supports a group mode in which two people sitting at different displays see and may scrawl on a replicated pair of windows at the same time.

Etch's primary function is to allow the user to scrawl handwritten messages by pressing the left mouse button and dragging the mouse. The right mouse button draws white, with the effect of erasing previously drawn pixels. The middle button draws straight lines between where the mouse button is depressed and where it is released. A "rubberband" line is redrawn continuously as the mouse moves, in order to assist the user in placing the line. The control-L key erases the screen, and the ESC key terminates the program.

Etch consists of a single procedure, main(av). *Etch* takes an optional command-line argument, the name of a remote display for communication purposes. If that argument is present, it is the first element in the list av; otherwise av is an empty list and its size (given by *av) is 0.

Etch begins by opening a square window with the line

```
w1 := open("etch", "x", "geometry=300,300") | stop("can't open window")
```

After the window is successfully opened, two additional bindings to the same window are created, one with invertible drawing operations and the other with opposite foreground and background colors by the calls

```
w2 := XBind(w1, "drawop=reverse")
w3 := XBind(w1, "reverse=on")
```

Note that the extra bindings are not really necessary, since w1's drawing operation and foreground and background colors can be modified using the XAttrib() function, but the use of XBind() shortens and simplifies code, and improves performance.

Following the opening of the window and creation of all bindings, the program goes into an event processing loop. In each step of the loop, an event is read with a call to XEvent(). An Icon case expression is used to handle the different kinds of events. Keyboard events consist of one character strings. The escape character "\e" simply breaks out of the event loop and the control-L character "\^l" simply calls XClearArea().

Null value tests are used to handle boundary cases where coordinates haven't yet been defined. The case clauses handling mouse drag event are:

```
&ldrag: {
    XDrawLine(w1, x1, y1, &x, &y)
    # left and right buttons use current position
    x1 := &x        # for subsequent operations
    y1 := &y
    }
&rdrag: {
    XDrawLine(w3, x1, y1, &x, &y)
    # left and right buttons use current position
    x1 := &x        # for subsequent operations
    y1 := &y
    }
&mdrag: {
    if /dragging then dragging := 1
    else {          # erase previous line, if any
        XDrawLine(w2, x1, y1, \x2, \y2)
        }
    x2 := &x
    y2 := &y
    XDrawLine(w2, x1, y1, x2, y2)
    }
```

The line drawing function requires that the mouse position at the time the button is pressed be available at the time the button is released. Extra variables are used to remember past events' positions so that lines can be erased and redrawn when the mouse is being dragged.

X-Icon Reference Manual

Table 1: Window Attributes

The following attributes are maintained on a per-window basis.

Window Attribute	Type	Source Name	Default
size, in rows and columns	integer	lines, columns	12x80
size, in pixels	integer	height, width	above
position on the screen	integer pair	pos	
position, x coordinate	integer	posx	
position, y coordinate	integer	posy	
size and position	geometry spec	geometry	
iconic state	window state	iconic	window
icon position	integer pair	iconpos	
icon image	image	iconimage	
icon label	string	iconlabel	
window label (title)	string	windowlabel	
cursor location (row,column)	integer	row, col	1,1
pointer (mouse) shape	mouse shape	pointer	X_cursor
pointer location (row,column)	integer	pointerrow, pointercol	
cursor location, in pixels	integer	x, y	
pointer location, in pixels	integer	pointerx, pointery	
device on which the window appears	device name	display	
display type	integer triple	visual	
display depth, in bits	integer	depth	
display height, in pixels	integer	displayheight	
display width, in pixels	integer	displaywidth	
visible cursor during raw input	switch	cursor	off
initial window contents	image file name	image	

Table 2: Graphics Context Attributes

The following attributes are maintained in graphics *contexts* that are independent of any particular window.

Context Attribute	Type	Source Name	Default
foreground and background color	color	fg, bg	
text font	font name	font	fixed
text font's max height, width	integer	fheight, fwidth	
text font leading	integer	leading	fheight
text font ascent, descent	integer	ascent, descent	fheight
drawing operation	logical op	drawop	copy
graphics fill style	fill style	fillstyle	solid
graphics line style	line style	linestyle	solid
graphics line width	integer	linewidth	1
reverse fg and bg colors	switch	reverse	off
clip rectangle position	integer	clipx, clipy	0
clip rectangle extent	integer	clipw, cliph	0
output translation	integer	dx, dy	0

Table 3: Attribute Types

The following table summarizes valid string values that may be assigned to the various attributes in the preceding tables.

Type	Values	Example
color	color name or rgb value	"red", "0,0,0", "#FFF"
device name	X display	"unix:0"
fill style	solid, stippled, opaquestippled	"solid"
font name	X font	"fixed"
geometry spec	*int*x*int*[+-]*int*[+-]*int*	"100x100+50+20"
image	file name	"flag.xpm"
integer	32 bit signed integers	"0"
integer pair	*int*,*int*	"0,0"
line style	solid, onoff, doubledash	"onoff"
logical op	(see Drawing operations)	"reverse"
mouse shape	(see Appendices)	"gumby"
string	any string	"hello"
switch	on, off	"off"
window state	normal, iconic, root	"iconic"

Built-in Functions

X-Icon adds the following built-in functions to Icon's repertiore. The functions are presented here using the conventions given in the Icon book [Gris90]. Arguments named x and y are pixel locations in zero-based integer graphics coordinates. Arguments named row and col are cursor locations in one-based integer text coordinates.

Functions with a first parameter named w default to &window and the window argument can be omitted in the default case. This may be viewed as analogous to write(). This defaulting behavior does not apply to functions with multiple window arguments.

---

close(w) : w — close window

close(w) closes a binding on a window. The window on the screen disappears after any associated binding is closed, but the window can still be written to and read/copied from until all open bindings are closed or the program terminates.

Errors: 103 s not string
See also: open(), XBind(), XUnbind()

---

open(s,"x",s_1,...,s_n) : w — open window

open(s, "x") opens a window for reading and writing with default text and graphic attributes. Non-default initial values for attributes can be supplied in the third and following arguments to open().

Errors: 103 s not string
See also: close(), XBind()

XActive() : w — produce active window

XActive() returns a window that has one or more events pending. If no window has an event pending, XActive() blocks and waits for an event to occur. To find an active window, it checks each window, starting with a different window on each call in order to avoid window "starvation". XActive() fails if no windows are open.

See also: XPending()

XAttrib(w,x_1,...,x_n) : s... generate or set attributes

XAttrib(w,x_1,...) retrieves and/or sets window and context attributes. It generates strings encoding the value of the attribute(s) in question. If called with more than two arguments, it prefixes each value by the attribute name and an equals sign (=). If x_i is a window, subsequent attributes apply to x_i. XAttrib() fails if it is given an invalid attribute name or if is is unable to set the attribute to the requested value.

Errors:	140	w not window
	109	x_i not string or window

XBg(w,x,g,b) : s background color

XBg(w,x,g,b) retrieves and/or sets background color by name, *rgb*, or mutable color value. With one argument, the background color is retrieved. With two arguments, the background color is set by a string color value (either a name or *rgb* encoded as a string) or integer mutable color value. With four arguments, the last three arguments are integers denoting the red, green, and blue components of the background color to set. Note that in Xlib the graphics context background color is distinct from the window background used whenever windows are cleared or enlarged. The window background is set during open() and may not be changed. XBg() fails if the background cannot be set to the requested color.

Errors:	140	w not window
	101	x or g or b not integer
	103	x not string (two arguments)

See also: XFg()

XBind(w_1,w_2,s_1,...,s_n) : w bind window to context

XBind(w_1,w_2) produces a new value that binds the window associated with w_1 to the graphics context associated with w_2. If w_2 is omitted, a new graphics context is created, with font and color characteristics defaulting to those present in w_1. If both window arguments are omitted, the new binding denotes a *pixmap* that is not associated with any window on the display at all. Additional string arguments specify initial attributes of the new binding, as in XAttrib(). XBind() fails if a display cannot be opened or if an attribute cannot be set to a requested value. &window is not a default for this function.

Errors:	140	w_1 or w_2 not window
	109	s_i not string

See also: XAttrib()

XClearArea(w,x,y,width,height) : w clear to window background

XClearArea(w,x,y,width,height) clears a rectangular area within the window to the window's background color. This is not the current background defined in the context, but rather it is the background color defined at window creation time in the call to open(), or white if there was none. If width is 0, the region cleared extends from x to the right side of the window. If height is 0, the region cleared extends from y to the bottom of the window.

Defaults:	x,y,width,height	0 (all)
Errors:	140	w not window
	101	x, y, width, or height not integer
See also:	XEraseArea()	

XClip(w,x,y,width,height) : w clip to rectangle

XClip(w,x,y,width,height) clips output to a rectangular area within the window. If width is 0, the clip region extends from x to the right side of the window. If height is 0, the clip region extends from y to the bottom of the window.

Defaults:	x,y,width,height	0 (all)
Errors:	140	w not window
	101	x, y, width, or height not integer

XColor(w,i,x,g,b) : w set mutable color

XColor(w,i) produces the current setting of mutable color i. XColor(w,i,x,g,b) sets the color map entry identified by i to the given color. XColor() fails in the case of an invalid color specification.

Error:	140	w not window
	101	i, x, g, or b not integer
	103	x not string (three arguments)
See also:	XNewColor()	

XCopyArea(w_1,w_2,x,y,width,height,x_2,y_2) : w copy area

XCopyArea(w_1,w_2,x,y,width,height,x_2,y_2) copies a rectangular region within w_1 defined by x,y,width,height to window w_2 at offset x_2,y_2. XCopyArea() returns w_1. &window is not a default for this function.

Defaults:	x, y, x_2, y_2	0
	width, height	0 (all of w_1)
Errors:	140	w_1 or w_2 not window
	101	x, y, width, height, x_2, or y_2 not integer

XDefault(w, program, option) : s query user preference

XDefault(w, program, option) returns the value of option for program as registered with the X resource manager. In typical use this supplies the program with a default value for window attribute option from a *program.option* entry in an .XDefaults file. XDefault() fails if no user preference for the specified option is available.

Errors:	140	w not window
	103	program or option not integer

XDrawArc(w,x,y,width,height,a_1,a_2,...) : w draw arc

XDrawArc(w,x,y,width,height,a_1,a_2,...) draws any number of arcs, including ellipses. Each is defined by six integer coordinates. x, y, width and height define a *bounding rectangle* around the arc; the center of the arc is the point ($\frac{x+width}{2}$, $\frac{y+height}{2}$). Angles are specified in 64th's of a degree. Angle a_1 is the start position of the arc, where 0 is the 3 o'clock position and the positive direction is *counter*-clockwise. Angle a_2 is not the end position, but rather specifies the direction and extent of the arc. When drawing a single arc, height defaults to width, producing a circular arc, a_1 defaults to 0 and a_2 defaults to 360 * 64 (a complete ellipse). Supplied angle values are treated modulo 360*64; larger values "wrap around" the circle or ellipse.

Default:	a_1	0
	a_2	360 * 64
Errors:	140	w not window
	101	argument not integer or bad argument count
See also:	XFillArc()	

XDrawCurve(w,x_1,y_1,...,x_n,y_n) : w — draw curve

XDrawCurve(w,x_1,y_1,...,x_n,y_n) draws a smooth curve connecting each x, y pair in the argument list. If the first and last point are the same, the curve is smooth and closed through that point.

Errors: 140 w not window
101 argument not integer or bad argument count

XDrawLine(w,x_1,y_1,...,x_n,y_n) : w — draw line

XDrawLine(w,x_1,y_1,...,x_n,y_n) draws lines between each adjacent x, y pair in the argument list.

Errors: 140 w not window
101 argument not integer or bad argument count
See also: XDrawSegment(), XFillPolygon()

XDrawPoint(w,x_1,y_1,...,x_n,y_n) : w — draw point

XDrawPoint(w,x_1,y_1,...,x_n,y_n) draws points.

Errors: 140 w not window
101 argument not integer or bad argument count

XDrawRectangle(w,x_1,y_1,$width_1$,$height_1$,...) : w — draw rectangle

XDrawRectangle(w,x_1,y_1,$width_1$,$height_1$,...) draws rectangles. The width and height arguments define the *perceived* size of the rectangle drawn, like the C version of this function. The actual rectangle drawn is width+1 pixels wide and height+1 pixels high.

Errors: 140 w not window
101 argument not integer or bad argument count
See also: XFillRectangle()

XDrawSegment(w,x_1,y_1,x_2,y_2,...) : w — draw line segment

XDrawSegment(w,x_1,y_1,x_2,y_2,...) draws line segments (alternating x, y pairs are connected).

Errors: 140 w not window
101 argument not integer or bad argument count

See also: XDrawLine()

XDrawString(w,x_1,y_1,s_1,...) : w — draw text

XDrawString(w, x, y, s) draws text s at coordinates (x, y). This function does not draw any background; only the characters' actual pixels are drawn. This function uses the drawop attribute, so it is possible to use "drawop=reverse" to draw erasable text. XDrawString() does not affect the text cursor position. Newlines present in s cause subsequent characters to be drawn starting at (x, current_y + leading), where x is the x supplied to the function, current_y is the y coordinate the newline would have been drawn on, and leading is the current leading associated with the binding.

Errors: 140 w not window
101 argument not integer or bad argument count

See also: XDrawLine()

XEraseArea(w,x,y,width,height) : w — erase to context background

XEraseArea(w,x,y,width,height) erases a rectangular area within the window to the context's current background color, the color defined by XBg() and/or the bg attribute of XAttrib(). If width is 0, the region cleared extends from x to the right side of the window. If height is 0, the region erased extends from y to the bottom of the window.

Default:	x,y,width,height	0 (all)
Errors:	140	w not window
	101	x, y, width, or height not integer
See also:	XClearArea()	

XEvent(w) : x read event

XEvent(w) retrieves the next event available for window w. If no events are available, XEvent() waits for the next event. Keystrokes are encoded as strings, while mouse events are encoded as integers. The retrieval of an event is accompanied by assignments to the keywords &x, &y, &row, and &col.

Error: 140 w not window
See also: XPending()

XFg(w,x,g,b) : s foreground color

XFg(w,x,g,b) retrieves and/or sets foreground by name, *rgb*, or mutable color value, similar to XBg(). XFg() fails if the foreground cannot be set to the requested color.

Errors: 140 w not window
101 x, g, or b not integer
103 x not string (two arguments)
See also: XBg()

XFillArc(w,x,y,width,height,a_1,a_2,...) : w draw filled arc

XFillArc(w,x,y,width,height,a_1,a_2,...) draws filled arcs, ellipses, and/or circles. Coordinates are as in XDrawArc() above (not off by one as in the Xlib version of this function).

Errors: 140 w not window
101 argument not integer or bad argument count
See also: XDrawArc()

XFillPolygon(w,x_1,y_1,...,x_n,y_n) : w draw filled polygon

XFillPolygon(w,x_1,y_1,...,x_n,y_n) draws a filled polygon. The beginning and ending points are connected if they are not the same.

Errors: 140 w not window
101 argument not integer or bad argument count
See also: XDrawLine()

XFillRectangle(w,x_1,y_1,$width_1$,$height_1$,...) : w draw filled rectangle

XFillRectangle(w,x_1,y_1,$width_1$,$height_1$,...) draws filled rectangles.

Errors: 140 w not window
101 argument not integer or bad argument count
See also: XDrawRectangle()

XFlush(w) : w flush window output

XFlush(w) flushes window output on window systems that buffer text and graphics output. Window output is automatically flushed whenever the program blocks on input. When this behavior is not adequate, a call to XFlush() sends all window output immediately, but does not wait for all commands to be received and acted upon. XFlush() is a no-op on window systems that do not buffer output.

Error: 140 w not window
See Also: XSync()

XFont(w,s) : s get/set font

XFont(w) produces the name of the current font. XFont(w,s) sets the window context's font to s and produces its name or fails if the font name is invalid. The valid font names are currently system-dependent. XFont() fails if the requested font name does not exist.

Errors: 140 w not window
103 s not string

XGotoRC(w,row,col) : w go to row, col

XGotoRC(w,row,col) is the same as XAttrib(w, "row=" || row, "col=" || col), coordinates are given in characters. The column calculation used by XGotoRC() assigns to each column the pixel width of the widest character in the current font. If the current font is of fixed width, this yields the usual interpretation.

Defaults: row, col 1
Errors: 140 w not window
101 row or col not integer
See also: XGotoXY()

XGotoXY(w,x,y) : w go to x, y

XGotoXY(w,x,y) is the same as XAttrib(w,"x=" || x,"y=" || y); coordinates are given in pixels.

Defaults:	x, y	0
Errors:	140	w not window
	101	x or y not integer
See also:	XGotoRC()	

XIconic(w,s) : w set iconic state

XIconic(w,s) requests that window w be reduced to an icon or expanded in size, depending on argument s. If s is "icon", the window is reduced to an icon. If s is "window", the window is expanded to full size. If s is omitted the window's state is queried; either "icon" or "window" is returned. This procedure is subject to the higher-level control of the X window manager and consistent behavior across all window managers cannot be guaranteed.

Errors:	140	w not window
	103	s not string

XIconImage(w,x) : w set icon image

XIconImage(w,x) sets window w's icon pixmap from either a preloaded pixmap image (if x is a window value) or a pixmap file (if x is a string). If x is omitted the call returns the icon (string file name) associated with the window, or the empty string if the pixmap was set from another window value. This procedure is subject to the higher-level control of the X window manager and consistent behavior across all window managers cannot be guaranteed.

Errors:	140	w not window
	103	x not window or string
See also:	XReadImage()	

XIconLabel(w,s) : w set icon label

XIconLabel(w,s) sets window w's icon label from string s. If s is omitted, the call returns the icon label associated with the window.

Errors:	140	w not window
	103	x not string
See also:	XWindowLabel()	

XMoveIcon(w,x,y) : w move icon

XMoveIcon(w,x,y) moves the icon associated with window w to coordinates x,y with respect to the upper lefthand corner of the screen. If w is iconified, the move is immediate, otherwise it specifies where w's icon will appear when w is iconified. This procedure is subject to the higher-level control of the X window manager and consistent behavior across all window managers cannot be guaranteed.

Defaults:	x, y	0
Errors:	140	w not window
	101	x or y not integer
See also:	XMoveWindow()	

XMoveWindow(w,x,y,width,height) : w move/resize window

XMoveWindow(w,x,y) moves window w to coordinates (x,y) with respect to the upper left-hand corner of the screen. XMoveWindow(w,x,y,width,height) moves and resizes the window. XMoveWindow() sends a *request* for a move to the window manager, but the actual move is subject to the window manager and may or may not take place.

Defaults:	x, y	0
Errors:	140	w not window
	101	x, y, width, or height not integer
See also:	XMoveIcon()	

XNewColor(w,x,g,b) : i allocate mutable color

XNewColor(w,x,g,b) allocates an entry in the color map and returns a small negative integer that can be used to specify this entry in calls to routines that take a color specification, such as XFg(). If x or x,g,b are specified, the entry is initialized to the given color. XNewColor() fails if it cannot allocate an entry.

Error:	140	w not window
	101	x, g, or b not integer
	103	x not string (two arguments)

See also: XEvent(), XFg(), and XColor()

XParseColor(w,s) : i, i, i generate rgb values from color name

XParseColor(w,s) generates three integers ranging from 0 to 65,535 denoting the rgb components from string color name s. XParseColor() also recognizes comma-separated decimal "*r*,*g*,*b*" component strings and standard X hexadecimal component "#*rgb*" where each of *r*, *g*, and *b* are one to four hexadecimal digits. XParseColor() fails if string s is not a valid color name.

Errors:	140	w not window
	103	s not string

XPending(w,x_1,...,x_n) : L produce event queue

XPending(w) produces the list of events waiting to be read from window w. If no events are available, this list is empty (its size is 0). XPending(w,x_1,...,x_n) adds x_1 through x_n to the end of w's pending list in guaranteed consecutive order.

Error:	140	w not window

See also: XEvent()

XPixel(w,x,y,width,height) : L produce window pixels

XPixel(w,x,y,width,height) produces pixel contents from a rectangular area within window w. width * height results are produced. Results are produced starting from the upper left corner and advancing down to the bottom of each column before the next one is visited. Pixels are returned in integer values; ordinary colors are encoded positive integers, while mutable colors are negative integers that were previously returned by XNewColor(). Ordinary colors are encoded with the most significant eight bits all zero, the next eight bits contain the red component, the next eight bits the green component, and the least significant eight bits contain the blue component. These eight-bit component values are the most-significant bits of regular X sixteen-bit color values.

Error: 140 w not window
101 x, y, width, or height not integer
See also: XEvent()
XNewColor()

XQueryPointer(w) : x, y produce mouse position

XQueryPointer(w) generates the x and y coordinates of the mouse relative to window w. If w is omitted, XQueryPointer() generates the x and y coordinates of the mouse relative to the upper left corner of the entire screen. XQueryPointer() fails if the supplied window argument denotes a pixmap created by XBind() instead of an ordinary window.

Error: 140 w not window
See also: XWarpPointer()

XReadImage(w,s,x,y) : i load image file

XReadImage(w,s,x,y) loads an image from the file named by s into window w at offset x,y. x and y are optional and default to 0,0. Two image formats are supported, the standard XBM bitmaps (black and white) and XPM pixmaps (color) [LeHo91]. If XReadImage() succeeds in reading the image file into window w, it returns either an integer 0 indicating no errors occurred or a nonzero integer indicating that one or more colors required by the image could not be obtained from the window system. XReadImage() fails if file s cannot be opened for reading or is an invalid file format.

Default: x, y 0
Errors: 140 w not window
103 s not string
101 x or y not integer

XSetStipple(w,width,bits,...) : w

define stipple pattern

XSetStipple(w,width,bits,...) defines a stipple pattern of width width. width must be a number between 1 and 32 inclusive. The least-significant width bits of each subsequent integer argument is interpreted as a row of the pattern. The pattern is used by the fill versions of the rectangle, arc and polygon drawing functions when the fillstyle attribute is stippled or opaquestippled.

Errors:	140	w not window
	101	width or bits not integer
	205	width out of range

XSync(w,s) : w

synchronize client and server

XSync(w,s) synchronizes the program with the server attached to window w on those window systems that employ a client-server model. Output to the window is flushed, and XSync() waits for a reply from the server indicating all output has been processed. If s is "yes", all events pending on w are discarded. XSynch() is a no-op on window systems that do not use a client-server model.

Errors: 140 w not window

See also: XFlush()

XTextWidth(w,s): i

compute text pixel width

XTextWidth(w,s) computes the pixel width of string s in the font currently defined for window w.

Errors:	140	w not window
	103	s not string

XUnbind(w) : w

release a binding

XUnbind(w) releases the binding associated with file w. Unlike close(), XUnbind() does *not* close the window unless all other bindings associated with that window are also closed.

Errors: 140 w not window

See also: XBind()

XWarpPointer(w,x,y) : w — move mouse pointer

XWarpPointer(w,x,y) moves the mouse pointer suddenly from one place to another on the screen. The x and y coordinates are relative to window w; they may be negative. If w is omitted, x and y are relative to the upper left corner of the screen.

Errors:	140	w not window
	101	x or y not integer

See also: XQueryPointer()

XWindowLabel(w,s) : w — set window label

XWindowLabel(w,s) sets window w's window label (also called the window's *title*) from string s. If s is omitted, the call returns the label associated with the window.

Errors:	140	w not window
	103	x not string

See also: XIconLabel()

XWriteImage(w,s,x,y,width,height) : w — save image file

XWriteImage(w,s,x,y,width,height) saves an image of dimensions width, height from window w at offset x, y to a file named s. x and y are optional and default to (0,0). width and height are optional and default to the entire window. The file is written in the color XPM pixmap format if the supplied file name argument ends in .xpm or the UNIX *compress(1)* program is called to produce a compressed format if the file name ends in .xpm.Z; otherwise, the image is written in the standard X Window bitmap format. XWriteImage() fails if s cannot be opened for writing.

Defaults:	x, y	0
	width, height	(all)
Errors:	140	w not window
	103	s not string
	101	x, y, width, or height not integer

Keywords

The keywords added for X-Icon are presented in alphabetical order. Several of these keywords are variables with assignment restricted to value of a particular type or types.

&col : i mouse location, text column

The value of &col is the mouse location in text columns at the time of the most recently processed event. &col is a variable and may be assigned any integer; when it is assigned, &x is updated to a corresponding pixel location in the current font on the default window.

&control : i control modifier flag

&control produces the null value if the control modifier key was pressed at the time of the most recently processed event, otherwise &control fails.

&interval : i time since last event

&interval produces the time between the most recently processed event and the event that preceeded it, in milliseconds.

&ldrag : i left mouse button drag

&ldrag produces the integer value that is returned by XEvent() to indicate a left button drag event.

&lpress : i left mouse button press

&lpress produces the integer value that is returned by XEvent() to indicate a left button press event.

&lrelease : i left mouse button release

&lrelease produces the integer value that is returned by XEvent() to indicate a left button release event.

&mdrag : i middle mouse button drag

&mdrag produces the integer value that is returned by XEvent() to indicate a middle button drag event.

&meta : i meta modifier flag

&meta produces the null value if the meta modifier key was pressed at the time of the most recently processed event, otherwise &meta fails.

&mpress : i middle mouse button press

&mpress produces the integer value that is returned by XEvent() to indicate a middle button press event.

&mrelease : i middle mouse button release

&mrelease produces the integer value that is returned by XEvent() to indicate a middle button release event.

&resize : i window resize event

&resize produces the integer value that is returned by XEvent() to indicate a window resize event.

&rdrag : i right mouse button drag

&rdrag produces the integer value that is returned by XEvent() to indicate a right button drag event.

&row : i mouse location, text row

The value of &row is the mouse location in text rows at the time of the most recently processed event. &row is a variable and may be assigned any integer; when it is assigned, &y is updated to a corresponding pixel location in the current font on the default window.

&rpress : i right mouse button press

&rpress produces the integer value that is returned by XEvent() to indicate a right button press event.

&rrelease : i right mouse button release

&rrelease produces the integer value that is returned by XEvent() to indicate a right button release event.

&shift : i shift modifier flag

&shift produces the null value if the shift modifier key was pressed at the time of the most recently processed event, otherwise &shift fails.

&window : i default window

The value of &window is the window argument default for all X-Icon functions. &window is a variable and may be assigned any value of type window.

&x : i mouse location, horizontal

The value of &x is the horizontal mouse location at the time of the most recently processed event. &x is a variable and may be assigned any integer; when it is assigned, &col is updated to a corresponding text coordinate in the current font on the default window.

&y : i mouse location, vertical

The value of &x is the vertical mouse location at the time of the most recently processed event. &y is a variable and may be assigned any integer; when it is assigned, &row is updated to a corresponding text coordinate in the current font on the default window.

Acknowledgements

The design of X-Icon has benefitted tremendously from group discussion both within and without the Icon Project and can be considered a group effort. In addition to the author, the primary contributors to X-Icon's design are Ralph Griswold and Gregg Townsend. After its initial conception, they made very substantial

improvements in X-Icon's usability and usefulness. Additional persons who contributed X-Icon design ideas include Ken Walker, Nick Kline, and Jon Lipp.

The implementation has been improved by several persons. Darren Merrill ported X-Icon to OS/2 Presentation Manager. Gregg Townsend implemented XDefault() and event modifier key and time interval support, and ferreted out many inconsistencies in the implementation. Sandra Miller wrote XDrawCurve(), implemented mutable colors, added icons to Icon, and made numerous other improvements. Steve Wampler and Bob Alexander contributed numerous suggestions, bug reports, and inspirational program examples.

References

[Flow89] Flowers, J. *X Logical Font Description Conventions* X Version 11, Release 5 edition. Software Distribution Center, M.I.T., Cambridge, MA, 1989.

[Gris90] Griswold, R. E. and Griswold, M. T. *The Icon Programming Language,* second edition. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[LeHo91] LeHors, A. *The X PixMap Format*. Groupe Bull, Koala Project, INRIA, France, 1991.

[Nye88] Nye, A., editor. *Xlib Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, California, 1988.

[Sche86] Scheifler, R. W. and Gettys, J. The X Window System. *ACM Transactions on Graphics*, 5:79–109, April 1986.

Appendix A: Sample Programs

Xm

```
#
# Name:   xm.icn
# Title:  simple X-Icon file browser
# Author: Clinton L. Jeffery
# Date:   September 23, 1991
#
procedure main(argv)
    if *argv = 0 then stop("usage: xm file")
    f := open(argv[1], "r") | stop("can't open ", argv[1])
    w := open(argv[1], "x") | stop("no window")
    L := [ ]
    every put(L, !f)
    close(f)
    base := 0
    repeat {
        XClearArea(w)
        XGotoRC(w, 1, 1)
        every i:= 0 to XAttrib(w, "lines") - 2 do {
            if i + base < *L then writes(w, L[i + base + 1])
            write(w)
            }
        XAttrib(w, "reverse=on")
        writes(w,"--More--(",
                ((100 > (base + XAttrib(w,"lines") - 2) * 100 / *L) | 100),
                "%)")
        XAttrib(w, "reverse=off")
        case reads(w, 1) of {
            "q": break
            " ": base := (*L > (base + XAttrib(w, "lines") - 2) | fail)
            "b": base := (0 < (base - XAttrib(w, "lines") + 2) | 0)
        }
    }
    close(w)
end
```

Bme

```
#
# Name:   bme.icn
# Title:  BitMap Editor
# Author: Clinton L. Jeffery
# Date:   Sept. 22, 1991
#
# An X-Icon bitmap editor.
#

procedure main(argv)
   WIDTH := HEIGHT := 32

   if argv[1] == "-geo" then {
      pop(argv)          # pop "-geo"
      argv[1] ? {
         WIDTH := integer(tab(many(&digits))) | stop("geo syntax")
         ="x" | stop("geo syntax")
         HEIGHT := integer(tab(0)) | stop("geo syntax")
         pop(argv)
         }
      }
   if (*argv > 0) & (f := open(s := (argv[1] | (argv[1] || ".xbm")))) then {
      close(f)
      w1 := open("BitMapEdit", "x", "width=" || (WIDTH * 10),
                 "height=" || (HEIGHT * 10), "pos=400,400") | stop("open")
      w2:= open("BitMap", "x", "width=" || WIDTH, "height=" || HEIGHT,
                 "pos=330,400", "image=" || s) | stop("open")
      # Construct magnified copy of bitmap
      every i := 0 to HEIGHT - 1 do {
         every j := 0 to WIDTH - 1 do {
            XCopyArea(w2, w1, j, i, 1, 1, j * 10, i * 10)
            XCopyArea(w2, w1, j, i, 1, 1, j * 10 + 1, i * 10)
            every k := 1 to 4 do {
               XCopyArea(w1, w1, j * 10, i * 10, 2, 1, j * 10 + k * 2, i * 10)
               }
            XCopyArea(w1, w1, j * 10, i * 10, 10, 1, j * 10, i * 10 + 1)
            every k := 1 to 4 do {
               XCopyArea(w1, w1, j * 10, i * 10, 10, 2, j * 10, i * 10 + k * 2)
               }
```

```
                }
            }
        }
    else {
        w1 := open("BitMapEdit", "x", "width=" || (WIDTH * 10),
                    "height=" || (HEIGHT * 10), "pos=400,400") |
                stop("open")
        w2:= open("BitMap", "x", "width=" || WIDTH, "height=" || HEIGHT, "pos=330,400") |
                stop("open")
        }

    XFg(w1, "black")
    every i := 0 to HEIGHT − 1 do
        every j := 0 to WIDTH − 1 do
            XDrawRectangle(w1, j * 10, i * 10, 10, 10)

    repeat {
        case e := XEvent(w1) of {
            "q": return
            "s": {
                /s := getfilename()
                XWriteImage(w2, s)
                }
            &lpress | &ldrag: {
                dot(w1, w2, &x / 10, &y / 10, 1)
                }
            &mpress | &mdrag: {
                dot(w1, w2, &x / 10, &y / 10)
                }
            }
        }
end

procedure dot(w1, w2, x, y, black)
    if \black then {
        XFg(w1, "black")
        XFg(w2, "black")
        }
    else {
        XFg(w1, "white")
        XFg(w2, "white")
```

```
        }
    XFillRectangle(w1, x * 10, y * 10, 10, 10)
    XDrawPoint(w2, x, y)
    XFg(w1, "black")
    if /black then XDrawRectangle(w1, x * 10, y * 10, 10, 10)
end

procedure getfilename()
    wprompt := open("Enter a filename to save the bitmap", "x", "font=12x24", "lines=1") |
                            stop("can't xprompt")
    rv := read(wprompt)
    close(wprompt)
    if not find(".xbm", rv) then rv ||:= ".xbm"
    return rv
end
```

Etch

```
#
# Name:     suetch.icn
# Title:    Simple Etch-A-Sketch
# Author:   Clinton L. Jeffery
# Date:     April 17, 1991
# See Also: etch.icn, a two-user version of this program
#
# An X-Icon drawing program.
#
# Dragging the left button draws black dots.
# The middle button draws a line from button press to the release point.
# The right button draws white dots.
# Control-L clears the screen.
# The Escape character terminates the program.
#

procedure main(av)
   #
   # open an etch window.
   # create a binding with reverse video for erasing.
   #
   w1 := open("etch", "x") | stop("can't open window")
   w2 := XBind(w1, "drawop=reverse")
   w3 := XBind(w1, "reverse=on")
   repeat {
      #
      # Wait for an available event.
      #
      e := XEvent(w)
      case e of {
         #
         # Mouse down events specify an (x1,y1) point for later drawing.
         # (x2,y2) is set to null; each down event starts a new draw command.
         #
         &lpress | &mpress | &rpress: {
            x1 := &x
            y1 := &y
            x2 := y2 := &null
            }
```

```
#
# Mouse up events obtain second point (x2,y2), and draw a line.
#
&lrelease: {
    XDrawLine(w1, x1, y1, &x, &y)
    }
&mrelease: {
    XDrawLine(w1, x1, y1, &x, &y)
    dragging := &null
    }
&rrelease: {
    XDrawLine(w3, x1, y1, &x, &y)
    }
#
# Drag events obtain a second point, (x2,y2), and draw a line
# If we are drawing points, we update (x1,y1); if we are
# drawing lines, we erase the "rubberband" line and draw a
# new one at each drag event; a permanent line will be drawn
# when the button comes up.
#
&ldrag: {
    XDrawLine(w1, x1, y1, &x, &y)
    # left and right buttons use current position
    x1 := &x        # for subsequent operations
    y1 := &y
    }
&rdrag: {
    XDrawLine(w3, x1, y1, &x, &y)
    # left and right buttons use current position
    x1 := &x        # for subsequent operations
    y1 := &y
    }
&mdrag: {
    if /dragging then dragging := 1
    else {          # erase previous line, if any
        XDrawLine(w2, x1, y1, \x2, \y2)
        }
    x2 := &x
    y2 := &y
    XDrawLine(w2, x1, y1, x2, y2)
    }
```

```
            "\014": {          # Control-L
                XClearArea(w1)
                }
            "\e": break
            }
        }
end
```

Appendix B: Notes for X Window System Users

Font names

Font names under X are server-dependent. Modern X servers use the X Logical Font Description Conventions [Flow89]. The application xlsfonts is used to list the fonts available on a given X server.

Colors

Valid color names are server-dependent. On many systems they live in a file named rgb.txt in the X library directory, often /usr/lib/X11.

Mouse Cursors

Under X, the mouse cursors available are taken from the standard X Window *cursor font*. Figure 2 shows these symbols and their associated string names. The valid names are derived from the corresponding C symbolic constants defined in <X11/cursorfont.h>. The values are derived from the C constants used in Xlib, shortened appropriately. Some of these strings have spaces in them, for example, "sb up arrow".

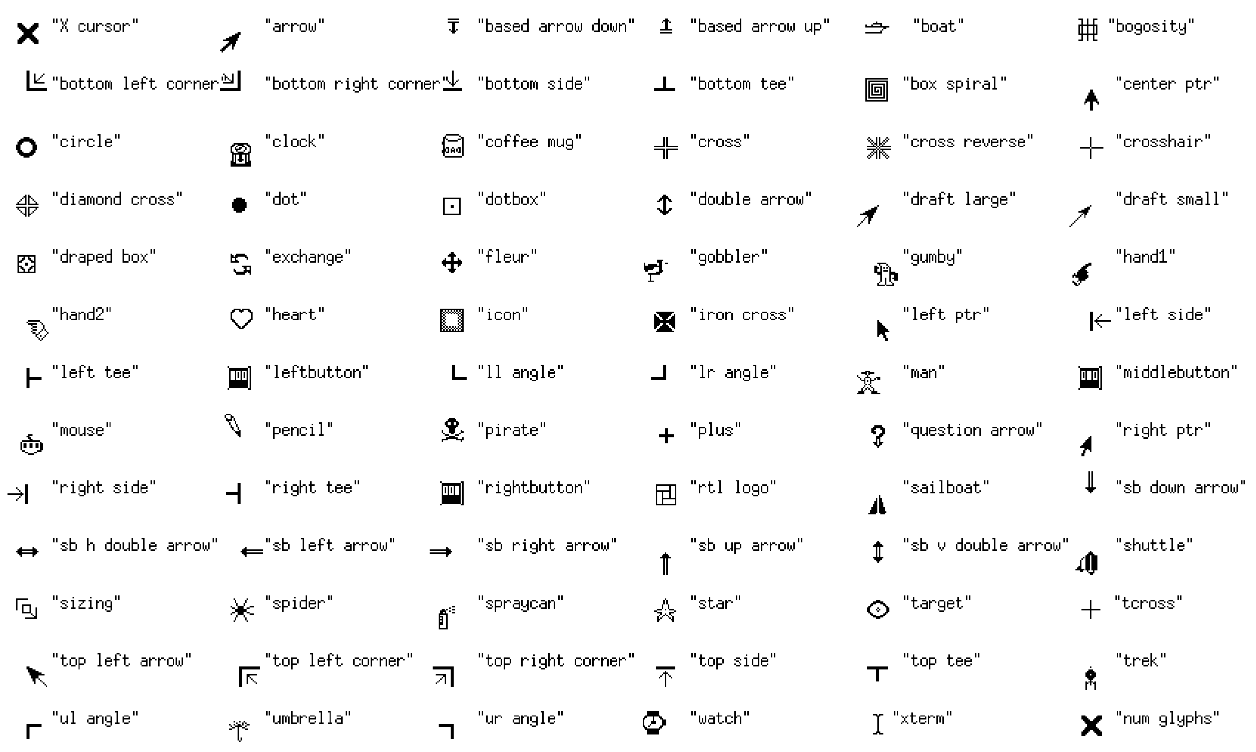

Figure 2: Mouse Cursors

Clients, servers, and multiple displays

The X Window System separates all graphics programs into two portions: the *client* and the *server*. The client is the actual application program, which computes values and makes various input and output requests for screen, keyboard, and mouse resources. X Window input and output requests are transmitted from the client, potentially across a network, to the main X program called the server. The server manages one collection of input and output devices (typically one or more screens, a keyboard, and a mouse) and arranges to share these resources among some number of client programs.

The separation of client and server has a number of implications. Because X is defined in terms of a network communication protocol, any operating system supporting that protocol can support X. In practice, client programs can be run on the fastest computer(s) available on the network, while the server's processor need only be powerful enough to manage the screen and input devices. This encourages resource sharing. On the other hand, the interprocess communication of the network protocol incurs a significant performance cost that is unnecessary in window systems that manage only locally-run programs. In addition, the client-server model creates an added level of complexity that makes X applications harder to write than those of many other graphics environments.

The display attribute takes as a value the name of an X Window server on which the window is to appear. This attribute can only be assigned to during the call to open() or XBind() in which the window is created. Most functions that involve multiple windows or shared resources, such as XBind(), only work on windows created on the same display.

Appendix C: Notes for OS/2 Presentation Manager Users

Font names

Fonts are specified by a comma-separated sequence of up to four fields supplying the font's family name, appearance, line weight, and size. Font family names available on all OS/2 systems include Times New Roman, Helvetica, Courier, and Symbol. Appearance may be condensed, expanded, or italic (currently only italic is supported). Line weight may be normal or bold. Font sizes are given in pixel height.

Not all fields need to be present and the order does not matter. For example, these two font specifications are equivalent: "font=Times New Roman, italic, 24" and "font=Times New Roman, 24, italic"

Some aliases are available for common font names:

Alias	Presentation Manager definition
fixed	System Monospaced
proportional	System Proportional
times	Times New Roman
helv	Helvetica

Color handling

The color names for OS/2's 16 base colors are:

black	blue	brown	cyan
darkblue	darkcyan	darkgray	darkgreen
darkpink	darkred	green	palegray
pink	red	white	yellow

Other defined color names in OS/2 X-Icon include:

beige	brown	chocolate	firebrick
gold	ivory	magenta	maroon
navy	orange	orchid	plum
purple	tan	violet	wheat

Under OS/2, rgb color values are composed of 8 bits per component. Color values from 1-255 are interpreted very differently under OS/2 than they are under X. OS/2 X-Icon accepts 16-bit X-style rgb values and uses the most significant 8 bits of 16-bit component value.

Mutable colors are not available. Mode reverse only works on the 16 base colors.

Stipples

XSetStipple() can take either a name specifying a pattern or a series of bits. In Presentation Manager, patterns are limited to an 8x8 patch. Extra bits are ignored and missing bits become 0.

The named patterns are:

blank	horizontalLines	solid	verticalLines
diagonal1	diagonal2	diagonal3	diagonal4
gray1	gray2	gray3	gray4
gray5	gray6	gray7	gray8

Console window

Writing to &output (or &errout) or reading from &input results in a console window appearing unless the streams are redirected. Trace output also goes to the console window.

Mouse pointer shapes are limited to:

arrow
clock
xterm (ibeam)

Miscellaneous

- At this time, only native (.bmp) bitmap file formats are supported for image read and write operations.
- XIconic (and the iconic attribute) also supports the value "fullscreen", which maximizes the window.
- Attributes "display" and "visual" are not supported.

Appendix D: Some X Keysyms

The table below shows some keysym integer values produced by XEvent() for special keys. These values were sampled on a Sun Sparcstation IPX running an MIT X11R5 server. The actual mapping between keys and keysyms is defined by the window system, and not by X-Icon.

Value	Key	Value	Key
65470	f1	65480	l1 (clash with f11)
65471	f2	65481	l2 (clash with f12)
65472	f3	65482	l3
65473	f4	65483	l4
65474	f5	65484	l5
65475	f6	65485	l6
65476	f7	65486	l7
65477	f8	65487	l8
65478	f9	65488	l9
65479	f10	65489	l10
65480	f11	65386	help
65481	f12		
65361	left	65490	pause
65362	up	65491	prsc
65363	right	65492	scroll lock
65364	down		
65496	home	65493	r4 (keypad =)
65498	pgup	65494	r5 (keypad /)
65500	middle (r11)	65495	r6 (keypad *)
65502	end	65312	compose
65504	pgdn	65379	insert

Appendix E: Event Queue Interpretation

X-Icon's event queue is an ordinary Icon list that can be accessed via XPending(). Each event is represented by three consecutive values on the list.

The first value is the event code: a string for a keypress event, or an integer for any other event. The next two values are integers, interpreted as a packed structure:

```
0000 0000 0000 0SMC XXXX XXXX XXXXXXXX (second value)
0EEE MMMM MMMM MMMM YYYY YYYY YYYYYYYY (third value)
```

The fields have these meanings:

`X...X`	&x: 16-bit signed x-coordinate value
`Y...Y`	&y: 16-bit signed y-coordinate value
`SMC`	&shift, &meta, &control modifier keys
`E...M`	&interval, interpreted as M * 16 ˆ E milliseconds
`0`	currently unused, should be zero

A malformed event queue error (error 143) is reported if an error is detected when trying to read the event queue. Possible causes for the error include an event queue containing fewer than three values, or second or third entries that are not integer values or are out of range.

Appendix F: Bugs and Deficiencies

- There is no means of creating *child* windows. All windows created by Icon are children of the root window and are managed separately by the window manager.
- There currently is no way to set many of the graphics attributes that may be available in the window system, e.g., fill rule, cap style, join style.
- X-Icon provides no way to directly access low level window system color facilities. This limits the effectiveness of the attribute drawop.
- Under X, event processing routines are poll-based rather than interrupt-based. Programs that block doing activities such as standard i/o reads are unable to handle even simple screen maintenance such as window movement. This is an artifact of the current implementation and the X Window system itself; the OS/2 version of X-Icon does not have this problem.
- The text cursor does not flash and is almost invisible when a large window full of text in a small font is displayed.