1 Introduction

X is an event driven protocol. An X program must take this into account in two ways: first, it must subscribe to specific events. Because there are so many events, and because most of the time, you're only interested in a subset of these events, you must explicitly subscribe to the events you're interested in. Secondly, once you've subscribed to events, you must respond to these events.

2 Event types

Events in X are stored in structs. An X event is actually a union, whose members are specific event types. When an event is received, it’s type member is checked, and then depending on what that is, a particular member of the union is looked at to get its values. First, let’s look at the definition of the X event, as found in the Xlib.h header file:

```c
typedef union _XEvent {
  int type; /* must not be changed; first element */
  XAnyEvent xany;
  XKeyEvent xkey;
  XButtonEvent xbutton;
  XMotionEvent xmotion;
  XCrossingEvent xcrossing;
  XFocusChangeEvent xfocus;
  XExposeEvent xexpose;
  XGraphicsExposeEvent xgraphicsexpose;
  XNoExposeEvent xnoexpose;
  XVisibilityEvent xvisibility;
  XCreateWindowEvent xcreatwindow;
  XDestroWindowEvent xdestroywindow;
  XUnmapEvent xunmap;
  XMapEvent xmap;
  XMapRequestEvent xmaprequest;
  XReparentEvent xreparent;
```
XConfigureEvent xconfigure;
XGravityEvent xgravity;
XResizeRequestEvent xresizerequest;
XConfigureRequestEvent xconfigurerequest;
XCirculateEvent xcirculate;
XCirculateRequestEvent xcirculaterequest;
XPropertyEvent xproperty;
XSelectionClearEvent xselectionclear;
XSelectionRequestEvent xselectionrequest;
XSelectionEvent xselection;
XColormapEvent xcolormap;
XClientMessageEvent xclient;
XMappingEvent xmapping;
XErrorEvent xerror;
XKeymapEvent xkeymap;
long pad[24];
} XEvent;

So, every time you get your hands on an event, you first check to see what the value of type is, usually with a switch-case statement. For example:

void ProcessEvent(XEvent e)
{
    switch(e.type)
    {
        case ButtonPress:
            ...
        case ConfigureNotify:
            ...
        case KeyPress:
            ...
    }

Once you’ve determined the type of event, you can dig into it’s sub-structure to get the values of the parameters you’re looking for. For example, suppose you want to know where the user clicked the mouse button. This information is sent via a XButtonEvent type event. It (the XButtonEvent) has data members x and y where the coordinates are stored. So, in your event handler, you might have something like:

void ProcessEvent(XEvent e)
{
    switch(e.type)
    {
        case ButtonPress:
            ButtonX=e.xbutton.x;
            ButtonY=e.xbutton.y;
            break;
    ...
Here, it is assumed you’ll do something with these values down the road. If you look at the XEvent union listed earlier, you’ll see the data member XButtonEvent xbutton. This is why in the code above you access the data member xbutton.

3 Subscribing to Events

Events are subscribed to by creating an event mask. Using the C or operator, you add events to the mask, and then tell the X client that a particular window on a particular display should be associated with this mask. So, how do you know which events you’re interested? For the most part, there are a handful that are required for most X programs. For each event in the XEvent union, there is a corresponding event mask. You should be aware that this is not a one-to-one relationship, as some masks correspond to more than one member in the union. In other words, for certain masks, you can expect to receive more than one event type. Here is a list of the event masks as defined in the file X.h.

```c
#define KeyPressMask (1L<<0)
#define KeyReleaseMask (1L<<1)
#define ButtonPressMask (1L<<2)
#define ButtonReleaseMask (1L<<3)
#define EnterWindowMask (1L<<4)
#define LeaveWindowMask (1L<<5)
#define PointerMotionMask (1L<<6)
#define PointerMotionHintMask (1L<<7)
#define Button1MotionMask (1L<<8)
#define Button2MotionMask (1L<<9)
#define Button3MotionMask (1L<<10)
#define Button4MotionMask (1L<<11)
#define Button5MotionMask (1L<<12)
#define ButtonMotionMask (1L<<13)
#define KeymapStateMask (1L<<14)
#define ExposureMask (1L<<15)
#define VisibilityChangeMask (1L<<16)
#define StructureNotifyMask (1L<<17)
#define ResizeRedirectMask (1L<<18)
#define SubstructureNotifyMask (1L<<19)
#define SubstructureRedirectMask (1L<<20)
#define FocusChangeMask (1L<<21)
#define PropertyChangeMask (1L<<22)
#define ColormapChangeMask (1L<<23)
#define OwnerGrabButtonMask (1L<<24)
```

The way you specify which events you’d like to subscribe to is by declaring an event mask (of type int) and by adding events to it. The or operator (which
is represented in C as —), sets bits to 1 corresponding to the events you’re interested in. For example, this code sets up a mask with several event types:

```c
int NotifyMask;
...
NotifyMask=0; //clear it of all events
NotifyMask=NotifyMask|=StructureNotifyMask;
NotifyMask=NotifyMask|KeyPressMask;
NotifyMask=NotifyMask|ButtonPressMask;
```

Now, we actually subscribe to events as follows:

```c
Display *display;
Window w;
...
//set up window and display
...
XSelectInput(display,w,Notify);
```

Now, everytime an event occurs in the window and display, specified by `w` and `display` in the code above, your program will be notified of it. The table below illustrates the commonly used events.

<table>
<thead>
<tr>
<th>Event Mask</th>
<th>Union member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyPressMask</td>
<td>xkey</td>
<td>Keyboard down</td>
</tr>
<tr>
<td>KeyReleaseMask</td>
<td>xkey</td>
<td>Keyboard up</td>
</tr>
<tr>
<td>ButtonPressMask</td>
<td>xbutton</td>
<td>Mouse button down</td>
</tr>
<tr>
<td>ButtonReleaseMask</td>
<td>xbutton</td>
<td>Mouse button up</td>
</tr>
<tr>
<td>PointerMotionMask</td>
<td>xmotion</td>
<td>Mouse pointer moved</td>
</tr>
<tr>
<td>ExposureMask</td>
<td>xexpose</td>
<td>Window is exposed</td>
</tr>
<tr>
<td>StructureNotifyMask</td>
<td>xconfigure</td>
<td>Window is changed</td>
</tr>
</tbody>
</table>

Here are the corresponding structs that you can access information from. So, when one of these events occurs, you may be interested in some of their data members.

```c
typedef struct {
int type; /* of event */
unsigned long serial; /* # of last request processed by server */
Bool send_event; /* true if this came from a SendEvent request */
Display *display; /* Display the event was read from */
Window window;    /* "event" window it is reported relative to */
Window root;      /* root window that the event occured on */
Window subwindow; /* child window */
Time time;        /* milliseconds */
int x, y;         /* pointer x, y coordinates in event window */
int x_root, y_root; /* coordinates relative to root */
unsigned int state; /* key or button mask */
```
typedef struct {
    int type; /* of event */
    unsigned long serial; /* # of last request processed by server */
    Bool send_event; /* true if this came from a SendEvent request */
    Display *display; /* Display the event was read from */
    Window window; /* "event" window it is reported relative to */
    Window root; /* root window that the event occurred on */
    Window subwindow; /* child window */
    Time time; /* milliseconds */
    int x, y; /* pointer x, y coordinates in event window */
    int x_root, y_root; /* coordinates relative to root */
    unsigned int state; /* key or button mask */
    unsigned int button; /* detail */
    Bool same_screen; /* same screen flag */
} XKeyEvent;
typedef XKeyEvent XKeyPressedEvent;
typedef XKeyEvent XKeyReleasedEvent;

typedef struct {
    int type; /* of event */
    unsigned long serial; /* # of last request processed by server */
    Bool send_event; /* true if this came from a SendEvent request */
    Display *display; /* Display the event was read from */
    Window window; /* "event" window it is reported relative to */
    Window root; /* root window that the event occurred on */
    Window subwindow; /* child window */
    Time time; /* milliseconds */
    int x, y; /* pointer x, y coordinates in event window */
    int x_root, y_root; /* coordinates relative to root */
    unsigned int state; /* key or button mask */
    char is_hint; /* detail */
    Bool same_screen; /* same screen flag */
} XButtonEvent;
typedef XButtonEvent XButtonPressedEvent;
typedef XButtonEvent XButtonReleasedEvent;

typedef struct {
    int type; /* of event */
    unsigned long serial; /* # of last request processed by server */
    Bool send_event; /* true if this came from a SendEvent request */
    Display *display; /* Display the event was read from */
    Window window; /* "event" window reported relative to */
    Window root; /* root window that the event occurred on */
    Window subwindow; /* child window */
    Time time; /* milliseconds */
    int x, y; /* pointer x, y coordinates in event window */
    int x_root, y_root; /* coordinates relative to root */
    unsigned int state; /* key or button mask */
    unsigned int button; /* detail */
    Bool same_screen; /* same screen flag */
} XMotionEvent;
typedef XMotionEvent XPointerMovedEvent;

typedef struct {
    int type;
    unsigned long serial; /* # of last request processed by server */
    Bool send_event; /* true if this came from a SendEvent request */
    Display *display; /* Display the event was read from */
4 Getting and Handling Events

There are couple of important functions used to actually process events. The first, XNextEvent fetches the next event on the queue (since events are constantly occurring, they are queued). A problem may arise, however, if there are no events pending. You may want to continue drawing your window, even if there are no events pending (for example, if you’re animating something). XNextEvent sits and waits for the next event, so your program flow might be interrupted if none are pending. The way you get around this is to see if there are any events before processing them. The easiest way to do this is to use the XPending function. Here’s an example event processor that uses both of these functions:

```c
void ProcessEvents()
{
    XEvent e;
    if(XPending(display))
    {
        XNextEvent(display,&e);
        switch(e.type)
        {
            case ButtonPress:
                ...
        }
    }
}
```

Notice how here, we first check to see if any events are pending before processing them. Also, as compared to earlier code, we’ve moved the grabbing of events
(via XNextEvent) to the event processor itself, rather than sending the event processor function an event grabbed in the main loop.

5 Drawing Commands

There are several draw functions available in X. The most common are listed below. The assumption in the examples is that previously in the code, a Display *, a Window and a GC have all been declared and initialized:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDrawPoint</td>
<td>Display, Window, GC, x, y</td>
</tr>
<tr>
<td>XDrawLine</td>
<td>Display, Window, GC, x1, y1, x2, y2</td>
</tr>
<tr>
<td>XDrawLines</td>
<td>Display, Window, GC, XPoint *, npoints, mode</td>
</tr>
<tr>
<td>XDrawSegments</td>
<td>Display, Window, GC, XSegment *, nsegments</td>
</tr>
<tr>
<td>XDrawRectangle</td>
<td>Display, Window, GC, x, y, width, height</td>
</tr>
<tr>
<td>XDrawArc</td>
<td>Display, Window, GC, x, y, width, height, ang1, ang2</td>
</tr>
<tr>
<td>XDrawString</td>
<td>Display, Window, GC, x, y, string, length</td>
</tr>
</tbody>
</table>

Note that XDrawLines and XDrawSegments use new X structures. They are defined as follows:

```c
typedef struct {
    short x, y;
} Xpoint;

typedef struct {
    short x1, y1, x2, y2;
} Xsegment;
```

XDrawLines assumes the line segments are connected, while XDrawSegments assumes they are not connected. You, of course, are responsible for allocating and deallocating the memory for these functions.

6 Sample Code

The code below responds to button down and button up events and draws a line from where the user first clicks to where the user releases the button.

```c
#include <X11/Xlib.h>
#include <math.h>
#include <stdio.h>

#define bool char
#define true 1
```
#define false 0

bool Running,ButtonDown;
Display *display;
Window w;
GC gc;
int width,height;
int X1,Y1,X2,Y2;

void ProcessEvent();
void Draw();

main()
{
int black,white,Notify;
display=XOpenDisplay(NULL);

black=BlackPixel(display,DefaultScreen(display));
white=WhitePixel(display,DefaultScreen(display));
Notify=0;
Notify|=StructureNotifyMask;
Notify|=KeyPressMask|ButtonPressMask|ButtonReleaseMask;
Notify|=PointerMotionMask;

//initialize variables
X1=Y1=X2=Y2=0;
ButtonDown=false;

w=XCreateSimpleWindow(display,DefaultRootWindow(display),0,0,
200,100,0,white,white);
if(display)
{
XSelectInput(display,w,Notify);
XMapWindow(display,w);
gc=XCreateGC(display,w,0,NULL);
XSetForeground(display,gc,black);
//wait for window map notify
Running=true;
while(Running)
{
ProcessEvent();
Draw();
}
}
}
void ProcessEvent()
{
XEvent e;
//if(XEventsQueued(display,QueuedAlready))
{
XNextEvent(display,&e);
switch(e.type)
{
    case ButtonPress:
        ButtonDown=true;
        X1=X2=e.xbutton.x;
        Y1=Y2=e.xbutton.y;
        break;
    case ButtonRelease:
        ButtonDown=false;
        break;
    case ConfigureNotify:
        width=e.xconfigure.width;
        height=e.xconfigure.height;
        printf("resized to %d x %d\n",width,height);
        break;
    case KeyPress:
        Running=false;
        break;
    case MotionNotify:
        if(ButtonDown)
        {
            X2=e.xmotion.x;
            Y2=e.xmotion.y;
        }
        break;
}
}
}

void Draw()
{
    char buffer[256];
    XClearWindow(display,w);
    XDrawLine(display,w,gc,X1,Y1,X2,Y2);
    sprintf(buffer,"X1: %d Y1: %d X2: %d Y2: %d length: %.2f", X1,Y1,X2,Y2,sqrt((Y2-Y1)*(Y2-Y1)+(X2-X1)*(X2-X1)));
}
XDrawString(display, w, gc, 2, height-4, buffer, strlen(buffer));
XFlush(display);
}