GNOME Window Manager Compliance - How to write a GNOME compliant Window Manager

GNOME Window Manager Compliance - How to write a GNOME compliant Window Manager

Table of Contents

This document provides quick and concise information for authors of Window Managers for the X Window System who wish to support the GNOME Desktop and its applications. You need to have a very good and detailed knowledge of the X Window System, Xlib, and how Applications and a Window Manager interact. A knowledge of ICCCM and experience in dealing with client interaction within a Window Manager framework is also assumed.

Chapter 1. Providing Client Information For The Window Manager

Section 1 - Detection of a GNOME compliant Window Manager

There is a single unambiguous way to detect if there currently is a GNOME compliant Window Manager running. It is the job of the Window Manager to set up a few things to make this possible. Using the following method it is also possible for applications to detect compliance by receiving an event when the Window Manager exits.

To do this the Window Manager should create a Window, that is a child of the root window. There is no need to map it, just create it. The Window Manager may reuse ANY window it has for this purpose - even if it is mapped, just as long as the window is never destroyed while the Window Manager is running.

Once the Window is created the Window Manager should set a property on the root window of the name **_WIN_SUPPORTING_WM_CHECK**, and type CARDINAL. The atom's data would be a CARDINAL that is the Window ID of the window that was created above. The window that was created would ALSO have this property set on it with the same values and type.

Example:

```
Display            *disp;
Window              root_window;
Atom                atom_set;
CARD32              val;
Window              win;

atom_set = XInternAtom(disp, "_WIN_SUPPORTING_WM_CHECK", False);
win = XCreateSimpleWindow(disp, root_window, -200, -200, 5, 5, 0, 0, 0);
val = win;
XChangeProperty(disp, root_window, atom_set, XA_CARDINAL, 32,
                PropModeReplace, (unsigned char *)&val, 1);
XChangeProperty(disp, win, atom_set, XA_CARDINAL, 32, PropModeReplace,
(unsigned char *)&val, 1);
```

Section 2 - Listing GNOME Window Manager Compliance

It is important to list which parts of GNOME Window Manager compliance are supported. This is done fairly easily by doing the following:

Create a property on the root window of the atom name **_WIN_PROTOCOLS**. This property contains a list(array)of atoms that are all the properties the Window Manager supports. These atoms are any number of the following:

- _WIN_LAYER
- _WIN_STATE
- _WIN_HINTS
- _WIN_APP_STATE
- _WIN_EXPANDED_SIZE
- _WIN_ICONS
- _WIN_WORKSPACE
- _WIN_WORKSPACE_COUNT
- _WIN_WORKSPACE_NAMES
- _WIN_CLIENT_LIST

If you list one of these properties then you support it and applications can expect information provided by, or accepted by the Window Manager to work.

```
Example:
  Display            *disp;
  Window              root_window;
  Atom                atom_set;
  Atom                list[10];

  atom_set = XInternAtom(disp, "_WIN_PROTOCOLS", False);
  list[0] = XInternAtom(disp, "_WIN_LAYER", False);
  list[1] = XInternAtom(disp, "_WIN_STATE", False);
  list[2] = XInternAtom(disp, "_WIN_HINTS", False);
  list[3] = XInternAtom(disp, "_WIN_APP_STATE", False);
  list[4] = XInternAtom(disp, "_WIN_EXPANDED_SIZE", False);
  list[5] = XInternAtom(disp, "_WIN_ICONS", False);
  list[6] = XInternAtom(disp, "_WIN_WORKSPACE", False);
  list[7] = XInternAtom(disp, "_WIN_WORKSPACE_COUNT", False);
  list[8] = XInternAtom(disp, "_WIN_WORKSPACE_NAMES", False);
  list[9] = XInternAtom(disp, "_WIN_CLIENT_LIST", False);
  XChangeProperty(disp, root_window, atom_set, XA_ATOM, 32, PropModeReplace,
                  (unsigned char *)list, 10);
```

Section 3 - Providing Shortcuts Managed Clients

As an aide in having external applications be able to list and access clients being managed by the Window Manager, a property should be set on the root window of the name **_WIN_CLIENT_LIST** which is an array of type CARDINAL. Each entry is the Window ID of a managed client. If the list of managed clients changes, clients are added or deleted, this list should be updated.

Example:

```
Display            *disp;
 Window              root_window;
 Atom                atom_set;
 Window             *wl;
 int                 num;

 atom_set = XInternAtom(disp, "_WIN_CLIENT_LIST", False);
 num = number_of_clients;
 wl = malloc(sizeof(Window) * num);
 /* Fill in array of window ID's */
 XChangeProperty(disp, root_window, atom_set, XA_CARDINAL, 32,
                 PropModeReplace, (unsigned char *)wl, num);
 if (wl)
   free(wl);
```

Section 3 - Providing Multiple/Virtual Desktop Information.

If your Window Manager supports the concept of Multiple/Virtual Desktops or Workspaces then you will definitely want to include it. This involves your Window Manager setting several properties on the root window.

First you should advertise how many Desktops your Window Manager supports. This is done by setting a property on the root window with the atom name _WIN_WORKSPACE_COUNT of type CARDINAL. The properties data is a 32-bit integer that is the number of Desktops your Window Manager currently supports. If you can add and delete desktops while running, you may change this property and its value whenever required. You should also set a property of the atom _WIN_WORKSPACE of type CARDINAL that contains the number of the currently active desktop (which is a number between 0 and the number advertised by _WIN_WORKSPACE_COUNT - 1). Whenever the active desktop changes, change this property.

Lastly you should set a property that is a list of strings called _WIN_WORKSPACE_NAMES that contains names for the desktops (the first string is the name of the first desktop, the second string is the second desktop, etc.). This will allow applications to know what the name of the desktop is too, possibly to display it.

Example:

```
Display            *disp;
Window              root_window;
Atom                atom_set;
XTextProperty       text;
int                 i, current_desk, number_of_desks;
char              **names, s[1024];
CARD32              val;

atom_set = XInternAtom(disp, "_WIN_WORKSPACE", False);
val = (CARD32) current_desk;
```

```
XChangeProperty(disp, root_window, atom_set, XA_CARDINAL, 32,
                PropModeReplace, (unsigned char *)&val, 1);
atom_set = XInternAtom(disp, "_WIN_WORKSPACE_COUNT", False);
val = (CARD32) number_of_desks;
XChangeProperty(disp, root_window, atom_set, XA_CARDINAL, 32,
                PropModeReplace, (unsigned char *)&val, 1);
atom_set = XInternAtom(disp, "_WIN_WORKSPACE_NAMES", False);
names = malloc(sizeof(char *) * number_of_desks);
for (i = 0; i < number_of_desks; i++)
  {
    snprintf(s, sizeof(s), "Desktop %i", i);
    names[i] = malloc(strlen(s) + 1);
    strcpy(names[i], s);
  }
if (XStringListToTextProperty(names, mode.numdesktops, &text))
  {
    XSetTextProperty(disp, root_window, &text, atom_set);
    XFree(text.value);
  }
for (i = 0; i < number_of_desks; i++)
  free(names[i]);
free(names);
```

Chapter 2. Reading State Requests From Clients

Section 1 - Initial Properties Set On Client Window

When a client first maps a window, before calling XMapWindow, it will set properties on the client window with certain atoms as their types. The property atoms set can be any or all of _WIN_LAYER, _WIN_STATE, _WIN_WORKSPACE, _WIN_EXPANDED_SIZE and _WIN_HINTS.

Each of these properties is of the type CARDINAL, and _WIN_EXPANDED_SIZE is an array of 4 CARDINAL's. For the _WIN_STATE and _WIN_HINTS properties, the bits set mean that state/property is desired by the client. The bitmask for _WIN_STATE is as follows:

```
#define WIN_STATE_STICKY          (1«0) /*everyone knows sticky*/
#define WIN_STATE_MINIMIZED       (1«1) /*Reserved - definition is unclear*/
#define WIN_STATE_MAXIMIZED_VERT  (1«2) /*window in maximized V state*/
#define WIN_STATE_MAXIMIZED_HORIZ (1«3) /*window in maximized H state*/
#define WIN_STATE_HIDDEN          (1«4) /*not on taskbar but window visible*/
#define WIN_STATE_SHADED          (1«5) /*shaded (MacOS / Afterstep style)*/
#define WIN_STATE_HID_WORKSPACE   (1«6) /*not on current desktop*/
#define WIN_STATE_HID_TRANSIENT   (1«7) /*owner of transient is hidden*/
#define WIN_STATE_FIXED_POSITION  (1«8) /*window is fixed in position even*/
#define WIN_STATE_ARRANGE_IGNORE  (1«9) /*ignore for auto arranging*/
```

These are a simple bitmasks - if the bit is set, that state is desired by the application. Once the application window has been mapped it is the responsibility of the Window Manager to set these properties to the current state of the Window whenever it changes states. If the window is unmapped the application is again responsible, if unmapped by the application.

The bitmask for _WIN_HINTS is as follows:

```
#define WIN_HINTS_SKIP_FOCUS      (1«0) /*"alt-tab" skips this win*/
#define WIN_HINTS_SKIP_WINLIST    (1«1) /*do not show in window list*/
#define WIN_HINTS_SKIP_TASKBAR    (1«2) /*do not show on taskbar*/
#define WIN_HINTS_GROUP_TRANSIENT (1«3) /*Reserved - definition is unclear*/
#define WIN_HINTS_FOCUS_ON_CLICK  (1«4) /*app only accepts focus if clicked*/
```

This is also a simple bitmask but only the application changes it, thus whenever this property changes the Window Manager should re-read it and honor any changes.

_WIN_WORKSPACE is a CARDINAL that is the Desktop number the app would like to be on. This desktop number is updated by the Window Manager after the window is mapped and until the window is unmapped by the application. The value for this property is simply the numeric for the desktop 0, being the first desktop available.

_WIN_LAYER is also a CARDINAL that is the stacking layer the application wishes to exist in. The values for this property are:

```
#define WIN_LAYER_DESKTOP                0
#define WIN_LAYER_BELOW                  2
#define WIN_LAYER_NORMAL                 4
#define WIN_LAYER_ONTOP                  6
```

```
#define WIN_LAYER_DOCK                   8
#define WIN_LAYER_ABOVE_DOCK             10
#define WIN_LAYER_MENU                   12
```

The application can choose one of these layers to exist in. It can also specify a layer other than the ones listed above if it wishes to exist between 2 layers. The layer remains constant and the window will always be arranged in stacking order between windows in the layers above and below its own layer. If the Window Manager changes the layer of an application it should change this property.

Section 2 - State Change Requests

After an application has mapped a window, it may wish to change its own state. To do this the client sends ClientMessages to the root window with information on how to change the application's state. Clients will send messages as follows:

```
Display             *disp;
Window               root, client_window;
XClientMessageEvent  xev;
CARD32                new_layer;

xev.type = ClientMessage;
xev.window = client_window;
xev.message_type = XInternAtom(disp, XA_WIN_LAYER, False);
xev.format = 32;
xev.data.l[0] = new_layer;
XSendEvent(disp, root, False, SubstructureNotifyMask, (XEvent *) &xev);

Display             *disp;
Window               root, client_window;
XClientMessageEvent  xev;
CARD32               mask_of_members_to_change, new_members;

xev.type = ClientMessage;
xev.window = client_window;
xev.message_type = XInternAtom(disp, XA_WIN_STATE, False);
xev.format = 32;
xev.data.l[0] = mask_of_members_to_change;
xev.data.l[1] = new_members;
XSendEvent(disp, root, False, SubstructureNotifyMask, (XEvent *) &xev);

Display             *disp;
Window               root, client_window;
XClientMessageEvent  xev;
CARD32               new_desktop_number;
```

If an application wishes to change the current active desktop it will send a client message to the root window as follows:

```
xev.type = ClientMessage;
xev.window = client_window;
xev.message_type = XInternAtom(disp, XA_WIN_WORKSPACE, False);
xev.format = 32;
xev.data.l[0] = new_desktop_number;
XSendEvent(disp, root, False, SubstructureNotifyMask, (XEvent *) &xev);
```

If the Window Manager picks up any of these ClientMessage events it should honor them.

Chapter 3. Desktop areas, button presses, and releases on the root window.

Section 1 - Button press and release forwarding for the desktop window.

X imposes the limitation that only 1 client can be selected for button presses on a window - this is due to the implicit grab nature of button press events in X. This poses a problem when more than one client wishes to select for these events on the same window - E.g., the root window, or in the case of a WM that has more than one root window (virtual root windows) any of these windows. The solution to this is to have the client that receives these events handle any of the events it is interested in, and then “proxy” or “pass on” any events it does not care about. The traditional model has always been that the WM selects for button presses on the desktop, it is only natural that it keep doing this BUT have a way of sending unwanted presses onto some other process(es) that may well be interested.

This is done as follows:

- 1. Set a property on the root window called _WIN_DESKTOP_BUTTON_PROXY. It is of the type cardinal - its value is the Window ID of another window that is not mapped and is created as an immediate child of the root window. This window also has this property set on it pointing to itself.

```
Display            *disp;
Window              root, bpress_win;
Atom                atom_set;
CARD32              val;

atom_set = XInternAtom(disp, "_WIN_DESKTOP_BUTTON_PROXY", False);
bpress_win = ECreateWindow(root, -80, -80, 24, 24, 0);
val = bpress_win;
XChangeProperty(disp, root, atom_set, XA_CARDINAL, 32,
                  PropModeReplace, (unsigned char *)&val, 1);
XChangeProperty(disp, bpress_win, atom_set, XA_CARDINAL, 32,
                  PropModeReplace, (unsigned char *)&val, 1);
```

- 2. Whenever the WM gets a button press or release event it can check the button on the mouse pressed, any modifiers, etc. - if the WM wants the event it can deal with it as per normal and not proxy it on - if the WM does not wish to do anything as a result of this event, then it should pass the event along like the following:

```
Display            *disp;
Window              bpress_win;
XEvent             *ev;

XUngrabPointer(disp, CurrentTime);
XSendEvent(disp, bpress_win, False, SubstructureNotifyMask, ev);
```

where **ev** is a pointer to the actual Button press or release event it receives from the X Server (retaining timestamp, original window ID, coordinates etc.)

NB - the **XUngrabPointer** is only required before proxying a press, not a release.

The WM should proxy both button press and release events. It should only proxy a release if it also proxied the press corresponding to that release.

It is the responsibility of any applications listening for these events (and as many applications as want to can since they are being sent under the guise of **SubstructureNotify** events), to handle grabbing the pointer again and handling all events for the mouse while pressed until release etc.

Section 2 - Desktop Areas as opposed to multiple desktops.

The best way to explain this is as follows. Desktops are completely geometrically disjoint workspaces. They have no geometric relevance to each other in terms of the client window plane. Desktop Areas have geometric relevance - they are next to, above, or below each other. The best examples are FVWM's desktops and virtual desktops - you can have multiple desktops that are disjoint and each desktop can be N x M screens in size - these N x M areas are what are termed "desktop areas" for the purposes of this document and the WM API.

If your WM supports both methods like FVMW, Enlightenment and possible others, you should use **_WIN_WORKSPACE** messages and atoms for the geometrically disjoint desktops - for geometrically arranged desktops you should use the **_WIN_AREA** messages and atoms. if you only support one of these it is preferable to use **_WIN_WORKSPACE** only.

The APi for **_WIN_AREA** is very similar to **_WIN_WORKSPACE**. To advertise the size of your areas (E.g., N x M screens in size) you set an atom on the root window as follows:

```
Display            *disp;
Window              root;
Atom                atom_set;
CARD32              val[2];

atom_set = XInternAtom(disp, "_WIN_AREA_COUNT", False);
val[0] = number_of_screens_horizontally;
val[1] = number_of_screens_vertically;
XChangeProperty(disp, root, atom_set, XA_CARDINAL, 32, PropModeReplace,
                (unsigned char *)val, 2);
```

To advertise which desktop area is currently the active one:

```
Display            *disp;
Window              root;
Atom                atom_set;
CARD32              val[2];

atom_set = XInternAtom(disp, "_WIN_AREA", False);
val[0] = current_active_area_x; /* starts at 0 */
val[1] = current_active_area_y; /* starts at 0 */
XChangeProperty(disp, root, atom_set, XA_CARDINAL, 32, PropModeReplace,
                (unsigned char *)val, 2);
```

If a client wishes to change what the current active area is they simply send a client message like:

```
Display            *disp;
Window              root;
XClientMessageEvent xev;

xev.type = ClientMessage;
xev.window = root;
xev.message_type = XInternAtom(disp, "_WIN_AREA", False);
xev.format = 32;
xev.data.l[0] = new_active_area_x;
xev.data.l[0] = new_active_area_y;
XSendEvent(disp, root, False, SubstructureNotifyMask, (XEvent *) &xev);
```

Chapter 4. The Future

Section 1 - What Else Is There?

There are currently a set of other hints available that are, at the current time, not essential and therefore not documented here. It is, however envisaged that they will be finalized and added to this document, but for now are not needed.