Linux Implementation: platform_linux.c

The Linux-specific implementation of the platform interface resides in `platform_linux.c`. The entire file is wrapped in a preprocessor conditional to ensure it is only compiled when targeting the Linux platform.

```
#if KPLATFORM_LINUX
//... Linux-specific implementation ...
#endif // KPLATFORM_LINUX
```

The X Window System: A Hybrid Approach

Windowing on Linux is typically handled by the **X Window System** (also known as X11), which operates on a client-server model. The core engine acts as a client that communicates with the X server to create windows and handle events.

There are two primary client libraries for this:

Xlib

An older, widely-known library. It is now considered largely deprecated.

XCB (X C Binding)

A more modern and efficient library that is the current standard.

This implementation uses a **hybrid approach**. It leverages Xlib for its straightforward display connection and key repeat management, while using the more robust XCB library for all other windowing and event-handling tasks.

Internal State & Dependencies

Internal State

To manage the Linux windowing system, a platform-specific `internal_state` structure is required. This is hidden from the rest of the engine via the opaque `void*` pointer in `platform_state`.

```
typedef struct internal_state {
  Display* display;
  xcb_connection_t* connection;
  xcb_window_t window;
  xcb_screen_t* screen;
  xcb_atom_t wm_protocols;
  xcb_atom_t wm_delete_win;
} internal_state;
```

- **Display* display**

  A pointer to the Xlib display connection. Used for initialization and key repeat settings.

- **xcb_connection_t* connection**

  The core XCB connection to the X server, used for most operations.

- **xcb_window_t window**

  The unique identifier (handle) for the application window.

- **xcb_screen_t* screen**

  A pointer to the primary screen data, containing information like resolution and default colors.

- **xcb_atom_t wm_protocols & wm_delete_win**

  Atoms used to register for and handle window manager events, such as the user clicking the close button.

Core Dependencies

On Debian-based systems, several development libraries are required to compile the platform layer. These provide the necessary headers for Xlib and XCB.

```
sudo apt-get install libx11-dev libxkbcommon-x11-dev libx11-xcb-dev
```

Other distributions require the equivalent X11/XCB development libraries.

Platform Startup: Connection & Window Creation

The `platform_startup` function on Linux follows a sequence of steps to establish a connection with the X server and create a window.

01. Connect to X Server

The initial connection is made using `XOpenDisplay` from the Xlib library. From this `Display` object, the more modern XCB connection is retrieved using `XGetXCBConnection`. This two-step process is the core of the hybrid approach.

02. Disable Key Repeats

`XAutoRepeatOff` is called to disable key repeats for the application.

> ⚠️ **Important:** This is a **global OS setting**. Forgetting to re-enable it on shutdown will leave key repeats disabled system-wide until the next reboot.

03. Prepare for Window Creation

Before a window can be created, the setup data is retrieved from the X server, and an iterator is used to find the primary screen. A unique ID for the new window is then allocated using `xcb_generate_id`.

04. Create the Window

`xcb_create_window` is called to create the window. Key parameters include:

- **event_mask**: Registers the types of events the application wants to listen for (e.g., `XCB_EVENT_MASK_KEY_PRESS`, `XCB_EVENT_MASK_POINTER_MOTION`).
- **value_list**: An array containing values corresponding to the value mask, such as the window's background color.

Platform Startup: Window Configuration

After the window is created, it must be configured and mapped to the screen.

Setting the Window Title

The window's title is not set during creation. A separate call to `xcb_change_property` is made, specifying the `XCB_ATOM_WM_NAME` atom to update the title text displayed by the window manager.

Handling Window Close Events

Unlike Win32, an X11 client must explicitly ask the window manager to send a notification when the user tries to close the window. This is a multi-step process:

→ 1. Get Atoms

`xcb_intern_atom` is called twice to get the unique identifiers for the strings "WM_PROTOCOLS" and "WM_DELETE_WINDOW". These are essentially registered strings that the X server understands.

→ 2. Set Protocol

`xcb_change_property` is called again. This time, it tells the window manager that our application will handle the `WM_DELETE_WINDOW` protocol.

Displaying the Window

The final steps are to make the window visible and ensure all commands are sent to the X server.

- `xcb_map_window` maps the window to the screen, making it visible.
- `xcb_flush` sends all buffered commands to the X server for execution. This is a blocking call that ensures the window creation process is complete before the function returns.

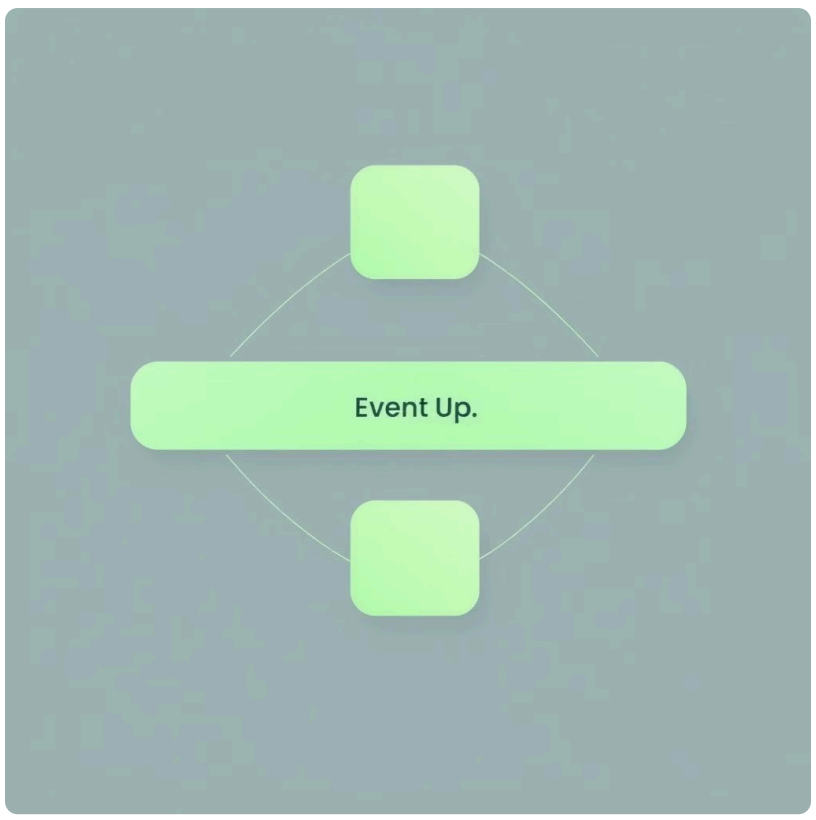
Message Pumping & Shutdown

Message Pumping

The `platform_pump_messages` function keeps the application interactive by processing events from the X server.

```
b8 platform_pump_messages(platform_state* plat_state) {
    internal_state *state = (internal_state*) plat_state->internal_state;
    xcb_generic_event_t* event;
    b8 quit_flagged = FALSE;

    while ((event = xcb_poll_for_event(state->connection))) {
        switch (event->response_type & ~0x80) {
            case XCB_CLIENT_MESSAGE:
                // Check if this is a window close event...
                if (((xcb_client_message_event_t*)event)->data.data32[0] ==
state->wm_delete_win) {
                    quit_flagged = TRUE;
                }
                break;
            // Handle other events (keyboard, mouse, etc.)...
        }
        free(event); // Free the event memory
    }
    return !quit_flagged;
}
```



It polls for events in a loop using `xcb_poll_for_event`.

A switch statement handles events based on their type. `XCB_CLIENT_MESSAGE` is checked against the saved `wm_delete_win` atom to detect when the user closes the window.

Crucially, the event object returned by `xcb_poll_for_event` is **dynamically allocated**. It must be released with `free()` after being handled to prevent memory leaks.

Platform Shutdown

The shutdown process is simple but critical for restoring system settings.

01

Re-enable Key Repeat

`XAutoRepeatOn` is called to re-enable the global key repeat setting.

02

Destroy Window Resources

`xcb_destroy_window` is called to release the window resources on the X server.

Timers, Sleep, and Console Output

High-Precision Time

The `platform_get_absolute_time` function provides a high-resolution timer.

- It uses `clock_gettime` with the `CLOCK_MONOTONIC` source, which represents the absolute time since system start and is not affected by changes to the system clock.
- It returns the time as a single `f64` (double-precision float) value representing the total number of seconds.

Platform Sleep

The `platform_sleep` implementation adapts to the available POSIX standard on the host system.

- If a modern standard is available (`_POSIX_C_SOURCE >= 199309L`), it uses `nanosleep()` for high-precision sleeping.
- Otherwise, it falls back to a combination of `sleep()` and `usleep()` for compatibility with older systems.

Colored Console Output

Both `platform_console_write` and `platform_console_write_error` use ANSI escape codes to produce colored text output in the terminal. The implementation is identical for both standard and error streams.

An array of color code strings is defined, corresponding to the different log levels (FATAL, ERROR, WARN, etc.). A single `printf` call is used to output the escape sequence, the color code, the message, and a final escape sequence to reset the color.

```
// Example of an ANSI colored printf
printf("\033[%sm%s\033[0m", color_string, message);
```