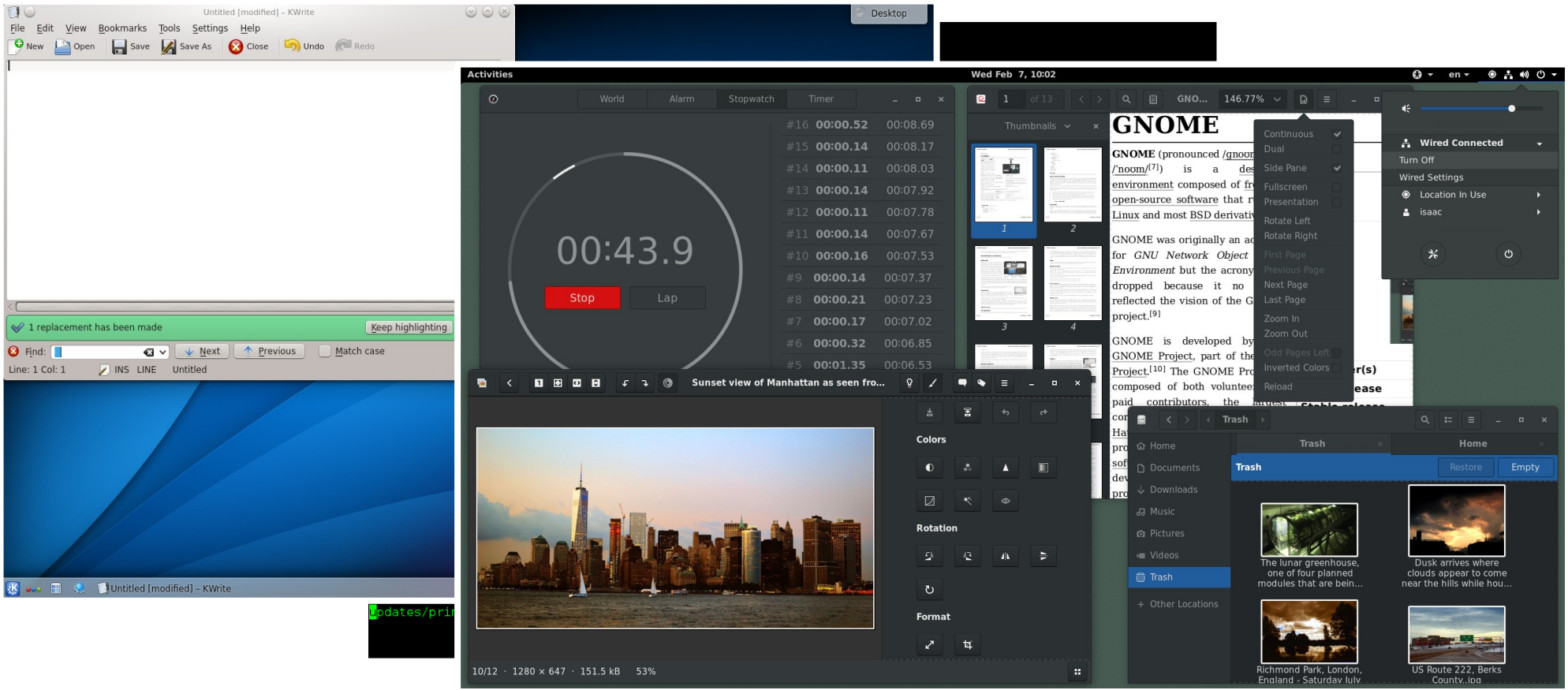
The Modern Linux Graphics Stack on Embedded Systems

Michael Tretter – m.tretter@pengutronix.de

User Interface for Linux Desktop



Desktop Environment / Window Manager

- Choose desktop environment (GNOME, KDE, ...)
- Install desktop environment
- Graphical user interface “just works”

But what about embedded systems?

Agenda

- Modern Linux Graphics Stack
- Graphics in Embedded Systems
- Weston for Embedded Graphics

Agenda

- **Modern Linux Graphics Stack**
- Graphics in Embedded Systems
- Weston for Embedded Graphics

Windowing System

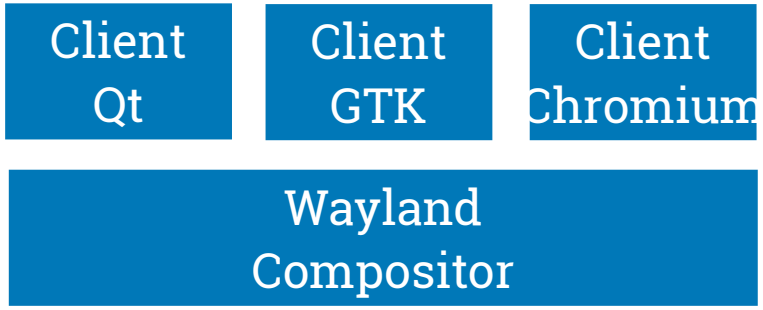
- Display server is central coordinator
- Applications talk to display server using specified protocols

Display Server

- Wayland compositor
- Compositor shell

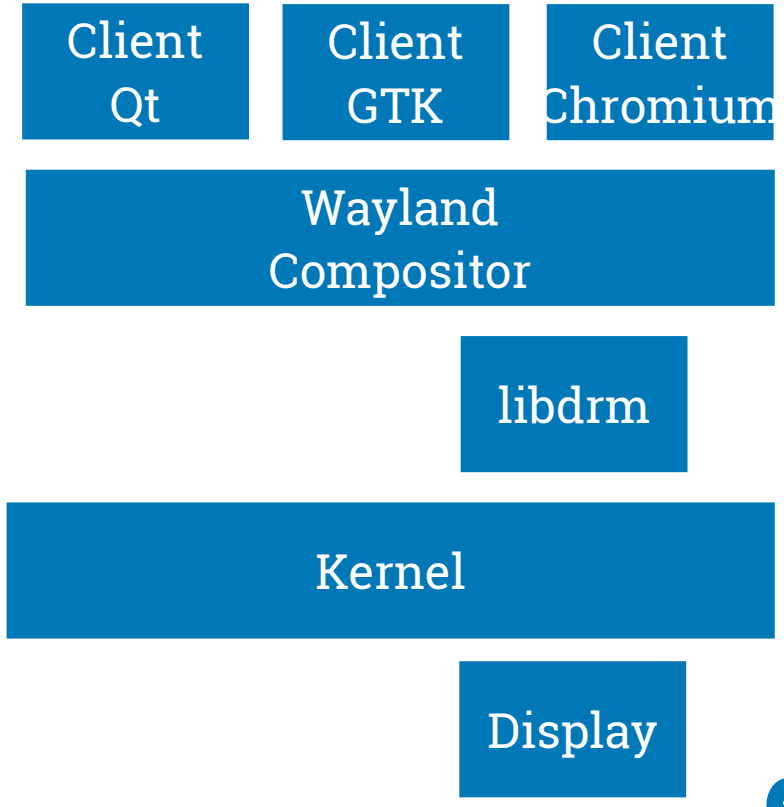
Wayland Client: xdg_shell Protocol

Description: “The xdg_wm_base interface is exposed as a global object enabling clients to turn their wl_surfaces into windows in a desktop environment. It defines the basic functionality needed for clients and the compositor to create windows that can be dragged, resized, maximized, etc, as well as creating transient windows such as popup menus.”



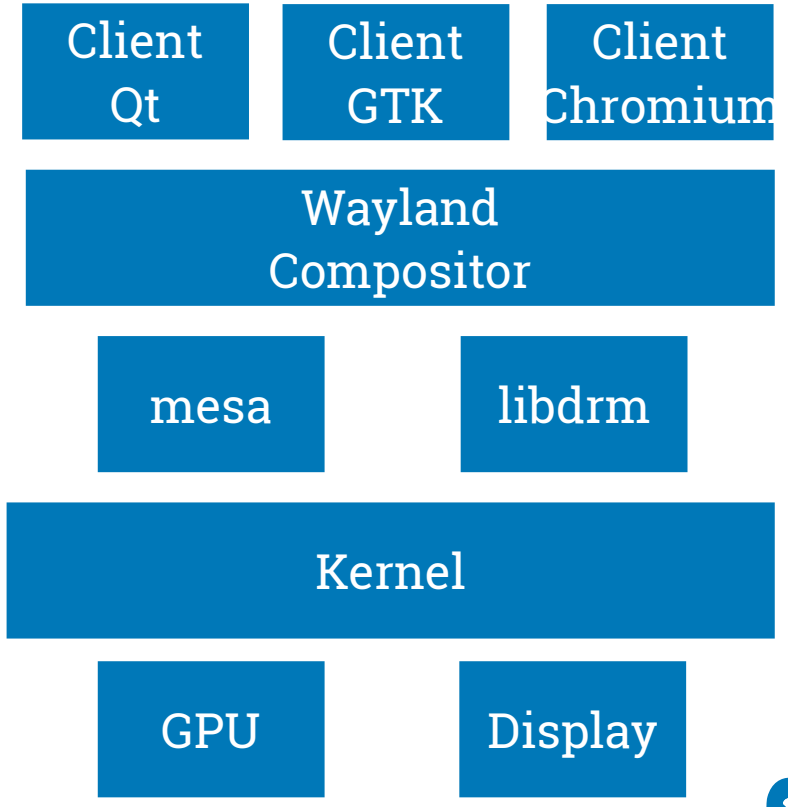
Display Output: DRM/KMS

- Framebuffer via DRM/KMS
- Composite client surfaces
- Flip framebuffer on output
- Kernel driver drives display



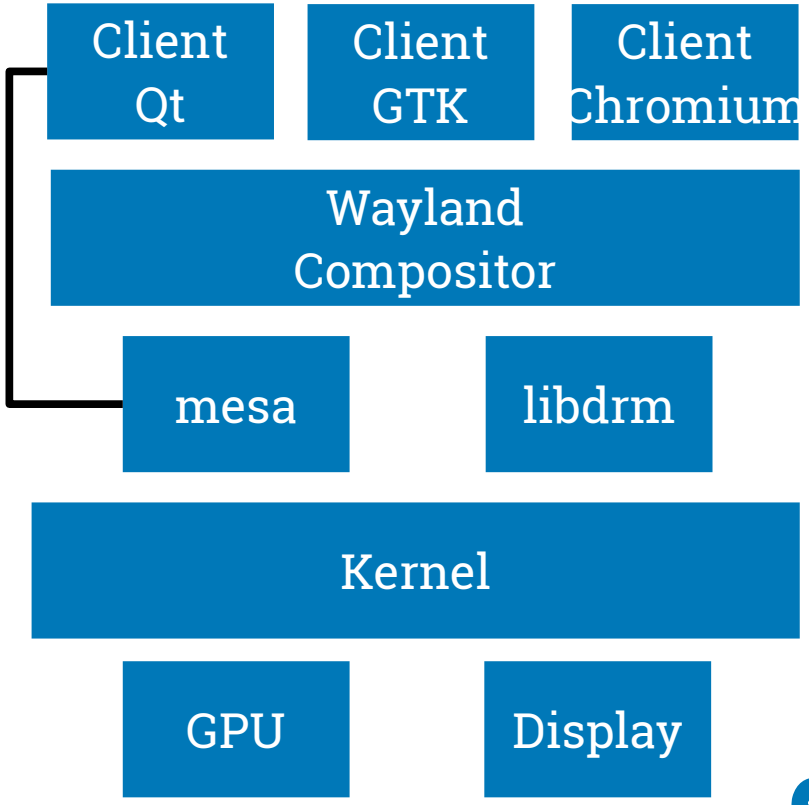
Surface Composition

- Compositor renders output frame from client surfaces
- Uses OpenGL for GPU acceleration
- Add animations or transparency



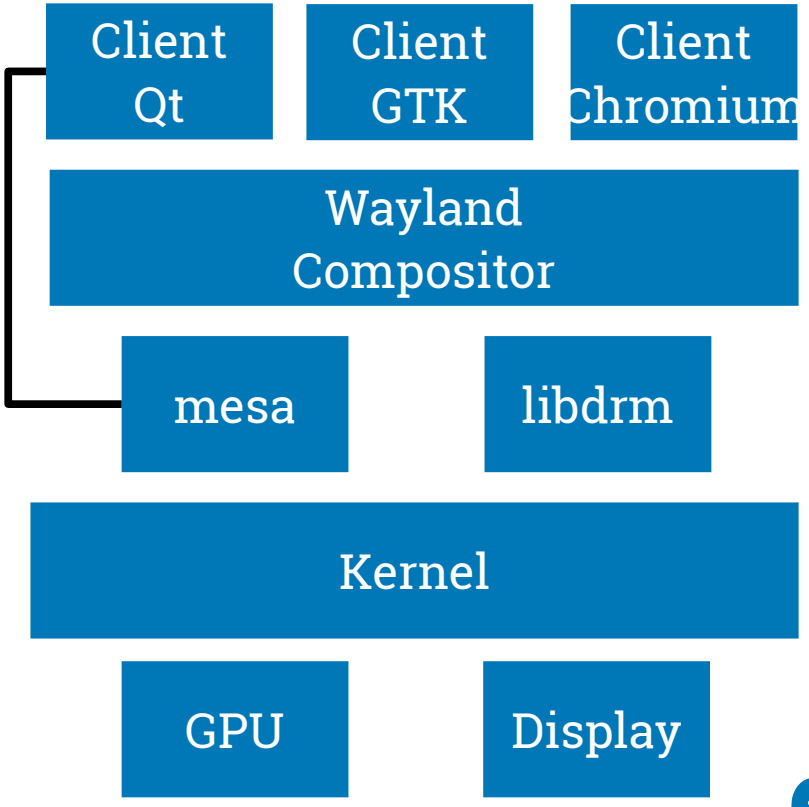
OpenGL for Wayland Clients

- Clients use their own buffer
- Applications directly use OpenGL for rendering



Graphics Stack Overview

- Application renders surface
- Compositor builds frame
- Display shows frame



Agenda

- Modern Linux Graphics Stack
- **Graphics in Embedded Systems**
- Weston for Embedded Graphics

What is so Special about Embedded?

- Limited hardware resources
- Limited memory bandwidth
- Limited (battery) power

Hardware accelerators for expensive functions

Graphics Hardware Features

- GPU for 3D hardware acceleration
- Display controllers with hardware overlays
- Hardware video encoders and decoders

- IP cores from different vendors in single SoC
- Pixel formats, tiling, alignment… **might** be compatible
- Custom hardware units for format translations

User Interface Development

- Application developers used to development on desktop
- Embedded systems have aforementioned peculiarities

Convenience of desktop user interface development for embedded systems?

Bridging the Gap

- Graphics stack must optimally use graphics hardware
- Drivers must provide proper hardware abstraction
- Userspace must understand and use the abstractions

Linux dma-buf Framework

- Share buffers without copy between multiple devices
- Driver export and import buffers as fd to user space
- Avoid copies between clients and compositor

Atomic Modesetting

- Perfect frames despite overlay planes
- Configure display and update atomically
- Composite surfaces in display controller

Videos and Pixel Formats

- YUV used in video encoding and produced by decoders
- Conversion necessary for GPU rendering
- Display controllers might directly support YUV format

Tiling and Format Modifiers

- Tiling reduces memory bandwidth usage
- Format modifiers describe buffer tiling
- Share compatible buffers between processing units

Let’s Look at the Userspace!

- APIs for passing tiled buffers between drivers are available
- Are compositors actually using these APIs?

Weston

Weston is the **reference implementation** of a Wayland compositor, as well as a useful environment in and of itself.

Out of the box, Weston provides a very basic desktop, or a full-featured environment for non-desktop uses such as automotive, **embedded**, in-flight, **industrial**, **kiosks**, set-top boxes and TVs. It also provides a library allowing other projects to build their own full-featured environments on top of Weston's core. (README.md of Weston)

Weston DRM Backend

With the DRM backend, weston runs without any underlying windowing system. The backend uses the **Linux KMS API** to detect connected monitors. [...] It is also possible to take advantage of **hardware** [...] **overlays**, when they exist and are functional. **Full-screen surfaces will be scanned out directly without compositing**, when possible. Hardware accelerated clients are supported via EGL. (man weston-drm)

compositor-drm.c: prepare planes

```
static int drm_plane_populate_formats(struct drm_plane *plane,
                      const drmModePlane *kplane,
                      const drmModeObjectProperties *props)
{
    blob = drmModeGetPropertyBlob(plane->backend->drm.fd, blob_id);
    for (i = 0; i < fmt_mod_blob->count_formats; i++) {
        plane->formats[i].format = blob_formats[i];
        plane->formats[i].modifiers = modifiers;
    }
}
```

compositor-drm.c: repaint

```
static int drm_output_repaint(struct weston_output *output_base, pixman_region32_t *damage, void *repaint_data) {
    drm_output_render(state, damage);
    scanout_state = drm_output_state_get_plane(state, output->scanout_plane);
    return 0;
}

static void drm_repaint_flush(struct weston_compositor *compositor, void *repaint_data) {
    struct drm_pending_state *pending_state = repaint_data;
    drm_pending_state_apply(pending_state);
}

static int drm_pending_state_apply_atomic(struct drm_pending_state *pending_state, enum drm_state_apply_mode mode) {
    ret = drmModeAtomicCommit(b->drm.fd, req, flags, b);
}
```

compositor-drm.c: plane assignment

```
static void drm_assign_planes(struct weston_output *output_base, void *repaint_data) {
    state = drm_output_propose_state(output_base, pending_state,
            DRM_OUTPUT_PROPOSE_STATE_PLANES_ONLY);
    if (!state) state = drm_output_propose_state(output_base, pending_state,
            DRM_OUTPUT_PROPOSE_STATE_MIXED);
    if (!state) state = drm_output_propose_state(output_base, pending_state,
            DRM_OUTPUT_PROPOSE_STATE_RENDERER_ONLY);
    wl_list_for_each(ev, &output_base->compositor->view_list, link) {
        wl_list_for_each(plane_state, &state->plane_list, link) {
            if (plane_state->ev == ev) {
                target_plane = plane_state->plane;
                break;
            }
        }
        if (target_plane) {
            weston_view_move_to_plane(ev, &target_plane->base);
        } else {
            weston_view_move_to_plane(ev, primary);
        }
    }
}
```

```
static struct drm_output_state *
drm_output_propose_state(struct weston_output *output_base,
            struct drm_pending_state *pending_state,
            enum drm_output_propose_state_mode mode)
{
    wl_list_for_each(ev, &output_base->compositor->view_list, link) {
        if (!ps && !force_renderer && !renderer_ok)
            ps = drm_output_prepare_scanout_view(state, ev, mode);
        if (!ps && !overlay_occluded && !force_renderer)
            ps = drm_output_prepare_overlay_view(state, ev, mode);
        if (ps)
            continue;
        pixman_region32_union(&renderer_region,
                &renderer_region, &clipped_view);
    }
    ret = drm_pending_state_test(state->pending_state);
}
```

compositor-drm.c: buffer import

```
static struct drm_fb *
drm_fb_get_from_view(struct drm_output_state *state, struct weston_view *ev) {
        dmabuf = linux_dmabuf_buffer_get(buffer->resource);
        if (dmabuf) {
                fb = drm_fb_get_from_dmabuf(dmabuf, b, is_opaque);
        } else {
                struct gbm_bo *bo;
                bo = gbm_bo_import(b->gbm, GBM_BO_IMPORT_WL_BUFFER,
                                   buffer->resource, GBM_BO_USE_SCANOUT);
                fb = drm_fb_get_from_bo(bo, b, is_opaque, BUFFER_CLIENT);
        }
}

static struct drm_fb *
drm_fb_get_from_bo(struct gbm_bo *bo, struct drm_backend *backend,
                   bool is_opaque, enum drm_fb_type type) {
        struct drm_fb *fb = gbm_bo_get_user_data(bo);
        fb->format = pixel_format_get_info(gbm_bo_get_format(bo));
        fb->modifier = gbm_bo_get_modifier(bo);
        return fb;
}
```

```
static struct drm_fb *
drm_fb_get_from_dmabuf(struct linux_dmabuf_buffer *dmabuf,
                       struct drm_backend *backend, bool is_opaque) {
        struct gbm_import_fd_modifier_data import_mod = {
                .width = dmabuf->attributes.width,
                .height = dmabuf->attributes.height,
                .format = dmabuf->attributes.format,
                .num_fds = dmabuf->attributes.n_planes,
                .modifier = dmabuf->attributes.modifier[0],
        };
        fb->bo = gbm_bo_import(backend→gbm,
                               GBM_BO_IMPORT_FD_MODIFIER,
                               &import_mod,
                               GBM_BO_USE_SCANOUT);
        fb->modifier = dmabuf->attributes.modifier[0];
        fb->format = pixel_format_get_info(dmabuf->attributes.format);
        return fb;
}
```

DRM Features Supported by Weston

Supported

- dma-buf import
- Atomic modeset
- Overlay planes
- Format modifiers

Unsupported

- Direct output of tiled buffers

Agenda

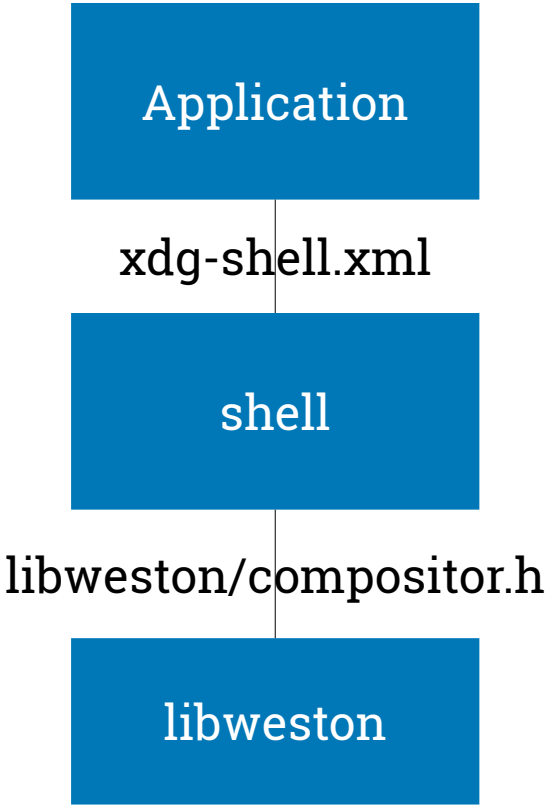
- Modern Linux Graphics Stack
- Graphics in Embedded Systems
- **Weston for Embedded Graphics**

Weston User Interface Development

- Compositor user interface defined by Weston shell
- Development on desktop using Wayland backend
- Testing with DRM backend on embedded system

Weston Shell

- Register shell in weston.ini
- Build shell as executable object file
- Implement xdg_shell protocol
- Make use of libweston



Weston Shell: Example

```
#include "xdg-shell-unstable-v6-server-protocol.h"
#include "libweston/compositor.h"
WL_EXPORT int wet_shell_init(struct weston_compositor *compositor, int *argc, char *argv[]) {
        /* ... */
        wl_global_create(display, &zxdg_shell_v6_interface, 1, data, xdg_shell_bind);
        /* ... */
}
static void surface_committed(struct weston_surface *surface, int32_t sx, int32_t sy) {
        /* ... */
        weston_compositor_schedule_repaint(compositor);
        /* ... */
}
```

Existing Weston Shells

- Desktop shell → Desktop
- IVI shell → In-Vehicle-Infotainment
- Fullscreen shell → Single fullscreen application

IVI Shell

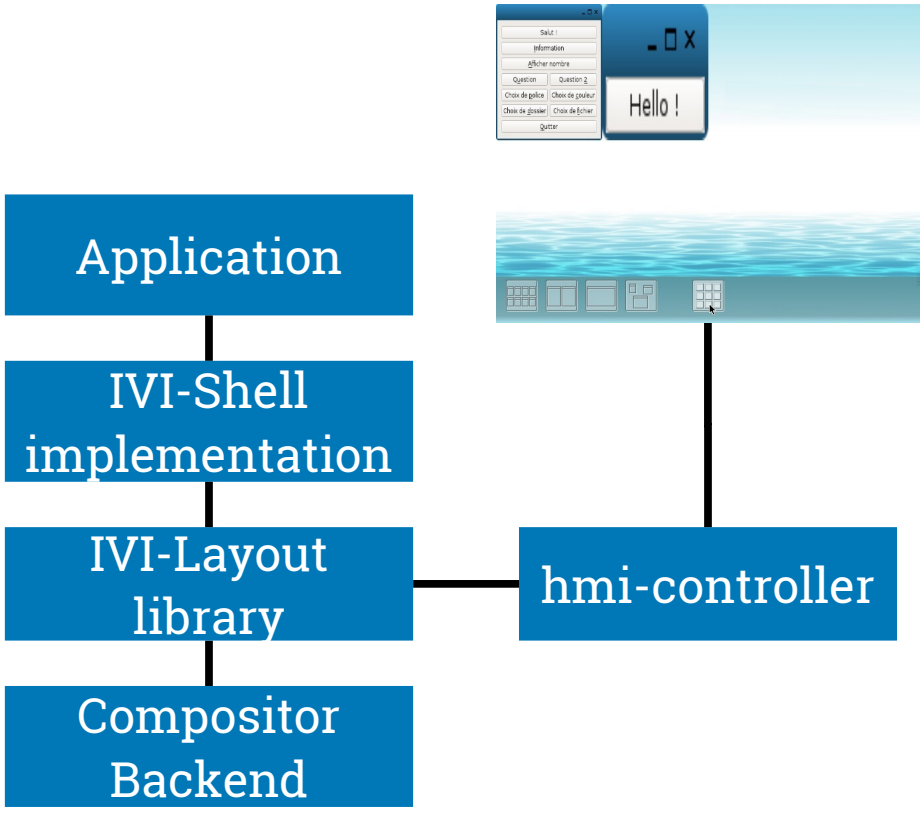
- Shell for embedded / HMI use case
- Compositor positions surfaces identified by ID
- Wayland protocol: ivi_application

xdg_shell not supported

IVI Shell with xdg_shell Support!

- Michael Teyfel: [PATCH weston v3 00/15] Desktop Protocol Support for IVI-Shell (Weston 4.0)
- https://lists.freedesktop.org/archives/wayland-devel/2018-April/037778.html

IVI Shell: Architecture



- IVI shell: Shell protocol
- IVI-Layout: layers and surfaces
- HMI-controller: user interface

Building a UI with IVI-Shell

- Replace hmi-controller with custom plugin
- Use ivi-layout API

```
#include <ivi-layout-export.h>
WL_EXPORT int wet_module_init(
        struct weston_compositor *compositor,
        int *argc, char *argv[])
{
        struct ivi_layout_interface *api;
        api = ivi_layout_get_api(compositor);
        /* ... */
        api->commit_changes();
        /* ... */
}
```

Alternatives to Weston?

- wlroots
- Qt Wayland Compositor

wlroots

- Modular Wayland compositor library
- wlr_layer_shell_unstable_v1 protocol
- Sway, Phosh, Rootston

- Overlay planes not supported
- Format modifiers not supported

Qt Wayland Compositor

- Qt module for developing display servers
- Provides QML and C++ APIs
- Well tested declarative UI technology

- Atomic modeset not supported
- Overlay planes not supported
- Format modifiers not supported

Open Questions

- Is the desktop environment useful for UI developers?
- Should the compositor really hide hardware complexity?
- How to decide between rendering and using overlay planes?
- Allow clients to control their surface position and layout?

Summary

- Linux graphics stack (Wayland, Mesa, DRM)
- Abstractions in DRM for peculiarities of embedded hardware
- How Weston uses the DRM interface
- How to build an embedded UI with Weston

Thank You

- Questions?
- Remarks?
- Opinions?
- Answers?

- Talk to me at the Pengutronix booth
- Write me an e-mail: m.tretter@pengutronix.de