FTF 2016
TECHNOLOGY FORUM

WAYLAND / WESTON ACCELERATION ON i.MX 6

FTF-DES-N1941

HUGO OSORNIO
GTEC GRAPHICS APPLICATIONS ENGINEER
FTF-DES-N1941
16 MAY 2016

AGENDA

- What is Wayland
- Why is it Useful
- What is Weston
- How to Add Wayland and Weston to Your BSP Using Yocto
- Overview of the Weston Source Code
- Review of Clients Code
- Creating Your First Weston Application
- Advanced Setups 1: Enabling a Multiple Seat System on Weston
- Advanced Setups 2: Enabling an Extended Desktop System on Weston
- Advanced Setups 3: XWayland

WHAT IS WAYLAND?

What is Wayland

- Wayland is a new display SERVER and COMPOSITION protocol.
- The protocol enables applications to allocate their own off-screen buffers and render their window contents directly, using hardware accelerated libraries like OpenGL ES, or high quality software implementations like Cairo.
- In the end what is needed is a way to present the resulting window surface for display, and a way to receive and arbitrate inputs among multiple clients.
- This is what Wayland provides by piecing together components already available in the ecosystem in a slightly different way.

What is Wayland

- The philosophy of Wayland is to provide clients with a away to manage windows and how their contents is displayed.

- Rendering is left to clients, and system wide memory management interfaces are used to pass buffer handles between the clients and the composition manager.

What is Wayland

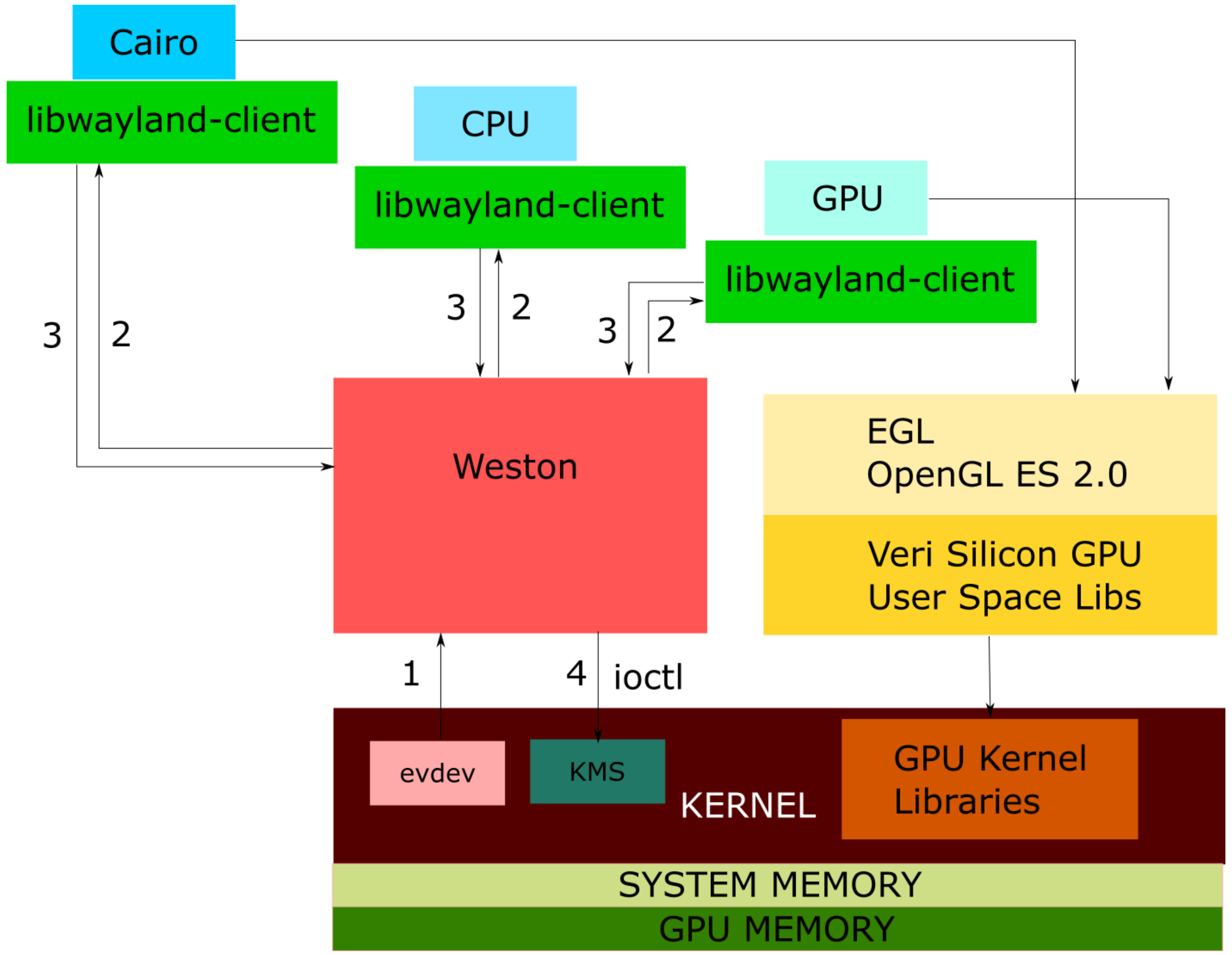
- The figure from the previous slide illustrates how Wayland Clients Interact with a Wayland server.
- Note that window management and composition are handled entirely on the server, significantly reducing complexity while marginally improving performance through reduced context switching.

What is Wayland – Overview of the Wayland Architecture

In this introductory phase, we will see how an input event is processed through Wayland:

1. The evdev module of the Linux kernel gets an event and sends it to the Wayland compositor.
2. The Wayland compositor looks through its scenegraph to determine which window should receive the event. The scenegraph corresponds to what's on screen and the Wayland compositor understands the transformations that it may have applied to the elements in the scenegraph. Thus, the Wayland compositor can pick the right window and transform the screen coordinates to window local coordinates, by applying the inverse transformations. The types of transformation that can be applied to a window is only restricted to what the compositor can do, as long as it can compute the inverse transformation for the input events.
3. As in the X case, when the client receives the event, it updates the UI in response. But in the Wayland case, the rendering happens in the client, and the client just sends a request to the compositor to indicate the region that was updated.
4. The Wayland compositor collects damage requests from its clients and then re-composites the screen. The compositor can then directly issue an ioctl to schedule a pageflip with KMS

What is Wayland



What is Not Wayland

- Wayland is ONLY a display server protocol, not a display server itself. Weston is the reference Wayland protocol implementation.

- Weston, covered in future slides, is used as one of the 3 main graphics backends throughout NXP i.MX 6 and i.MX 8 (coming soon!!) platforms.

WHY IS WAYLAND IMPORTANT?

Why is Wayland Important?

- Its MAIN reason of existence is that X11 is dated and already phasing out, X11 is over 30 years old and includes much functionality on the server side that is not needed anymore.
- X.Org, however, is still using it and doing a great job working around some of the core issues of X11, adding modular extensions, created DRI2 interface for instance. But then again, they are working around those issues.
- Wayland is restarting everything, instead of revolving around a 30 years old architecture, it allowed SW enginers to have a clean slate, to get rid of everything that they did not need. The main objective of this protocol is to happily compose and that’s it!

Why is Wayland Important?

- A lot of popular Linux distros, like Ubuntu rely or relied on the X windowing systems.
- On these systems, the X stack has grown to encompass functionality that does not necessarily belong to a display server like PCI resource management, display configuration management, direct rendering and memory management, heck the X server can even DRAW content as fonts to the final framebuffer!
- This causes high complexity on the end system and some redundant layers.

Why is Wayland Important?

- What interests most of us software dwellers, is the relative simplicity between a Wayland based server against a X11 server, in this session we will look at the source code of a reference implementation, across all levels, a client, the window manager, the final compositor, and you will leave with a solid idea of how to meddle with all those layers!

WHAT IS WESTON?

What is Weston?

- Weston is the reference implementation of a Wayland compositor.

- To delight of those C++ haters it is written completely in C and is, for the time being, exclusively for Linux due to dependence on certain features, such as Linux's kernel mode setting and udev, which have not yet been implemented in other Unix-like operating systems.

- When running on the Linux Kernel, handling of the input hardware relies on evdev.

- It contains a plugin system, external "shells" for WM/dock/etc, and Weston supports X clients. Clients are responsible for the drawing of their window borders and their decorations.

- For rendering, Weston can use OpenGL ES or software, but also allows the implementer to create its own composition scheme using HW layer managers and blitters.

What is Weston?

- Weston is focused to be a minimalist display server and compositor, its job is to compose the final frame buffer that will be displayed, arbitrate inputs and manage windows.

- Wayland and Weston do NOT have blocking calls nor spinlocks as with the X11 backend.

- IT WON’T DRAW, X11 had the capabilities to add window decorations, even printing fonts!, Weston and Wayland will never meddle with your buffers in such horrible ways!!

CREATING YOUR WESTON / WAYLAND BSP IMAGE

Yocto

- The Yocto Project is an open-source collaboration focused on embedded Linux® OS development.
- For more information regarding Yocto Project, see the Yocto Project page: www.yoctoproject.org. There are several documents on the Yocto Project home page that describe in detail how to use the system.
- Explaining Yocto requires a session of its own, I will only give you the steps you need to get your Yocto image up and running and start screwing up some Weston source code

Yocto Steps

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
  build-essential chrpath socat libsdl1.2-dev
$ sudo apt-get install libsdl1.2-dev xterm sed cvs subversion coreutils texi2html \
    docbook-utils python-pysqlite2 help2man make gcc g++ desktop-file-utils \
    libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf automake groff curl lzop asciidoc
$ sudo apt-get install u-boot-tools
$ mkdir ~/bin
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
$ export PATH=~/bin:$PATH
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
$ git config –list
$ mkdir fsl-release-bsp
$ cd fsl-release-bsp
$ repo init -u git://git.freescale.com/imx/fsl-arm-yocto-bsp.git -b imx-3.14.52-1.1.0_ga
$ repo sync
```

Yocto Steps 2 – Almost There

- After all those steps, your rig will be able to build Yocto and also have all the recipes to do so, we only need now to specify the image details we want to build, like the board and the graphics backend, which in this case will be **WAYLAND,** to do so execute the following steps:
- $ DISTRO=fsl-imx-**wayland** MACHINE=imx6qsabresd source fsl-setup-release.sh -b build-wayland
- $ bitbake fsl-image-gui
- After executing this step (and waiting 2 hours of intense compilation), you will have a wayland based i.MX6Q image where you will be able to play with all the knowledge we provided here.

Yocto – Waiting the Compile

While you wait those 2 hours, you can be curious and open the recipes for Weston, you will see the dependencies for the package to be compiled:

Open the file:

```
vi <YOCTO_DIR>/sources/meta-fsl-bsp-release/imx/meta-bsp/recipes-graphics/wayland/weston_1.9.0.bb
```

```
DEPENDS = "libxkbcommon gdk-pixbuf pixman cairo glib-2.0 jpeg"
DEPENDS += "wayland virtual/egl pango libinput"
```

Additionally to those dependencies, Weston requires the OpenGL ES and G2D libraries to do the final composition, they are not included here as they are already part of the default image.

Yocto – Locating the Weston and Source Codes

Once your image has been properly generated, you will find the Weston source codes in:

<YOUR YOCTODIR>/build-wayland/tmp/work/cortexa9hf-vfp-neon-mx6qdl-poky-linux-gnueabi/weston/1.9.0-r0/weston-1.9.0

Most of the code we will parse in this session will be located there

OVERVIEW OF THE WESTON SOURCE CODE

Overview of the Weston Source Code – Entry Point

The entrance point to the Weston compositor is located in the main.c file at the main function:

Upon startup weston will create the master wl_display structure, the clients running under the same compositor will only need to call the wl_display_connect to reference to this display structure.

During the display creation all the required lists and global objects will be created.

Client applications will create listeners to these global objects, an udev structure is created within these steps.

After this, you create the backend that you will use, this is the backend that the compositor will use to create the surfaces and communicate with the framebuffer, in the case of NXP, we use the fbdev backend compositor.

Once the backend is set, you proceed to create the compositor, once the compostitor is created, weston has everything it needs to render to the display.

In the backend_init function, the compositor will, after the compositor is created, launch your shell and the shell will be the master application that will render the background, panels, screen saver. Other clients will render on top of this surface.

Overview of the Weston Source Code – Backend

NXP Weston Implementation relies on the FBDEV implementation to interface with the final framebuffers

You will find this implementation on the src/compositor-fbdev.c file, here you will also be able to see how are the rendering surfaces created using the GPU provided functions to create rendering surfaces, look in the fbdev_output_create function. There you will notice how the final surfaces are created and mapped using the fbCreateWindow function.

This window (representing the final FB) will be then passed to the low level compositor so it can draw the client subsurfaces within it.

Overview of the Weston Source Code – Compositor Options

NXP Weston implementation provides the user 2 types of compositor. One relies on OpenGL ES 2.0 and the other on the GC320 or G2D, a dedicated HW piece capable of blitting and compositing.

Within the same src/ folder you will find the compositor.c file, which provides the common routines for the compositor and the gl-renderer.c and gal2d-renderer.c files, in those files the composition magic takes place, this is where the Weston-buffer coming from the client applications is received, as well as the master frame buffer, and the compositor blits those clients buffers into the final FB.

Overview of the Weston Source Code – Desktop Shell

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Weston comes with several flavors of Graphical shells, the one loaded by default is the desktop-shell.

The desktop shell consists on both, a protocol plugin and a client, that enables the window management functionality of the Weston implementation, It will handle requests such as window resizes, window movements, window animations, focus, priorities.

You can check the available shell functions under:

TODO

Overview of the Weston Source Code – Desktop Shell – Supported Functions

Desktop shell is like a modern X desktop environment, concentrating on traditional keyboard and mouse user interfaces and the familiar desktop-like window management.

Desktop shell consists of the shell plugin desktop-shell.so and the special client weston-desktop-shell which provides the wallpaper, panel, and screen locking dialog.

TODO LIST THE PLUGIN FUNCTIONS

SAMPLE CLIENT 1 WESTON INFO

Sample Client 1 – Weston Info

- Simple client that extracts information of each interface available on the current Wayland/Weston implementation.

- Interfaces are used for interacting with the server. Each interface provides requests, events, and errors

- Find below some of NXP’s Wayland Implementation you have the following interfaces, for the full list, execute the Weston-info sample.

- **interface: 'wl_compositor', version: 3, name: 1**
- Defined in Wayland
- Singleton object to communicate with the compositor, you can use it to create surfaces and regions.

Sample Client 1 – Weston Info

- **interface: 'wl_subcompositor', version: 1, name: 2**
- Defined in Wayland
- Interface to compose sub-surfaces

- **interface: 'wl_shm', version: 1, name: 6**
- Defined in Wayland
- Provides support for shared memory pools
- Clients can create wl_shm_pool objects using the create_pool request.
- At connection setup time, the wl_shm object emits one or more format events to inform clients about the valid pixel formats that can be used for buffers.

Sample Client 1 – Weston Info

- interface: 'wl_viv', version: 1, name: 7
- interface: 'wl_viv', version: 1, name: 8
- Defined in Vivante GPU source code
- Vivante extensions to create buffers in GPU memory
-
- interface: 'wl_output', version: 2, name: 9
- Defined in Wayland
- An output describes part of the compositor geometry, the actual Framebuffer size.

Sample Client 1 – Weston Info

- interface: 'wl_shell', version: 1, name: 14
- Defined in Weston
- This interface is implemented by servers that provide desktop-style user interfaces. It allows clients to associate a wl_shell_surface with a basic surface.

- interface: 'xdg_shell', version: 1, name: 15
- Defined in Weston
- xdg_shell allows clients to turn a wl_surface into a "real window" which can be dragged, resized, stacked, and moved around by the user.

- interface: 'desktop_shell', version: 3, name: 16
- Traditional user interfaces can rely on this interface to define the foundations of typical desktops. Currently it's possible to set up background, panels and locking surfaces.

Sample Client 1 – Weston Info – How does the code work

- Open the file weston-info.c
- 1.- The first step (Common among EVERY Wayland Client) is connect the display.
- **info.display = wl_display_connect(NULL);**
-
- 2.- Afterwards it will create a wl_list called infos. (What will it be populated with the avaible interfaces from the server.)
- **wl_list_init(&info.infos);**

- 3.- Create a registry object that allows us to list the global objects from the compositor
- **info.registry = wl_display_get_registry(info.display);**

Sample Client 1 – Weston Info – How does the code work

- 4.- Next we add a listener to the global objects that the compositor throws at us.
- **wl_registry_add_listener(info.registry, &registry_listener, &info);**

- 5.- The roundtrip will dispatch the requests to the server and its answer, and make sure the server and client execute the functions they registered. In this case, the registry_listener will execute the global_handler. This function is executed once every time the server sends a global event to its registered clients.
- **wl_display_roundtrip(info.display);**

- 6.- Upon execution of the global handler new requests will be added to the queue that also require to be dispatched. The wl_list declared on Step 2 is populated with the info of each interface and then printed.

SAMPLE CLIENT 2 WESTON FULLSCREEN

Sample Client 2 – Preface, the client common setup

- Many of the client samples on Weston use the Window.c helper file for the common interactions with the server.
- Display create on window.c takes care of the following major steps:
- 1.- wl_display_connect.
- 2.- wl_display_get_registry.
- 3.- wl_registry_add_listener
- When adding a listener to the registry, the callback that you register is called for every global object that the server reports, on this function it will add listeners for each interface. Output, Seats, Surface events among others.

Sample Client 2 – Preface, the client common setup

- The next big function in the Window.c file is: window_create
- 4.- wl_compositor_create_surface (Creates a compositor surface)
- 5.- adds a listerner to that surface so you detect changes on the surface.
- 6.- Sets the surface to be a xdg_shell surface, which will enable it to be moved, poped, scaled, according to the xdg interface and adds a listener for the surface.
- 7.- When the first frame is rendered, it will create the surface in the GPU memory area where the content will be finally rendered.
-
- Each client that uses this helper file will register its own functions that will be executed after the listeners.

Sample Client 2 – Weston-Fullscreen

- This example uses the xdg_shell interface to the compositor to handle the resizing of a window and setting it as full screen.
- We will focus only on the parts that we will ACTUALLY use when creating our simple application.
- Files to open on this sample:
- CLIENT SIDE:
- clients/fullscreen.c
- clients/window.c
- xdg-shell-client-protocol.h
- SERVER SIDE
- xdg-shell-server-protocol.h
- desktop-shell/shell.c
- compositor.c

Sample Client 2 – Weston-Fullscreen

- Go to the main function:
- //STEPS 1, 2, 3 and 4
- d = display_create(&argc, argv);
- //STEPS 4,5,6
- fullscreen.window = window_create(d);
-
- Once the common setup is ready, the client will register its own listener functions, those are functions that will take place:

Sample Client 2 – Weston-Fullscreen

- STEP 7
- widget_set_button_handler(fullscreen.widget, button_handler);
- Adds the button_handler function to the mouse button handler of window.c, this function will be called everytime the mouse button is pressed.
- If the left button is pressed, it will move the window using the following functions:
- window_move
- -> xdg_surface_move called in window.c, defined in xdg-shell-client-protocol.h
- -> wl_proxy_marshal assembles a request with index 5, XDG_SURFACE_MOVE, called in xdg-shell-client-protocol.h

Sample Client 2 – Weston-Fullscreen

- xdg_surface_interface defined in xdg-shell-protocol.h specifies the interface that receives the request and executes it with the function.
- -> xdg_surface_move (implementation) shell.c

Sample Client 2 – Weston-Fullscreen

- STEP 8
- widget_set_motion_handler(fullscreen.widget, motion_handler);

- Every time the pointer moves within a window, the server will send a pointer motion event to the client that has the pointer living within. On the case of this handler, every time the pointer moves, the widget needs to be redrawn as a new position may requires new data to be printed.

- Check the redraw_handler,every change needs to be redrawn.

Sample Client 2 – Weston-Fullscreen

- STEP 9
- widget_set_enter_handler(fullscreen.widget, enter_handler);

- Everytime the pointer ENTERS the client surface, it will send an event. Our client schedules a redraw as soon as the pointer enters its surface so it can report immediately its position.

Sample Client 2 – Weston-Fullscreen

- STEP 10
- widget_set_touch_down_handler(fullscreen.widget, touch_handler);

- Adds the touch_handler function to the touch handler of window.c, this function will be called everytime the touch screen reports a touch down event.

- If a touch down event is received, it will move the window using the following functions:
- window_move
- -> xdg_surface_move called in window.c, defined in xdg-shell-client-protocol.h
- -> wl_proxy_marshal assembles a request with index 5, XDG_SURFACE_MOVE, called in xdg-shell-client-protocol.h

Sample Client 2 – Weston-Fullscreen

- xdg_surface_interface defined in xdg-shell-protocol.h specifies the interface that receives the request and executes it with the function.
- -> xdg_surface_move (implementation) shell.c

Sample Client 2 – Weston-Fullscreen

- STEP 11
- window_set_key_handler(fullscreen.window, key_handler);
- Everytime the server detects a key, it will send the client currently in focus an event of a key stroke. Fullscreen client will capture the following keys:
- S: modifies scale
- wl_surface_set_buffer_scale, called in **window.c** defined in **wayland-client-protocol.h**
- assembles a message with an ID of 8 to the wl_surface WL_SURFACE_SET_BUFFER_SCALE
- wl_surface receives the message, the function wth index 8 is set_buffer_scale, this is defined in **wayland-server-protocol.h**
- The Surface interface is defined in the **compositor.c** file, the request will execute the surface_set_buffer_scale function that tells the compositor to scale the buffer.

Sample Client 2 – Weston-Fullscreen

- F: sends the compositor a full screen request via the xdg interface.
- window_set_fullscreen(window, fullscreen->fullscreen);
- xdg_surface_set_fullscreen, called in **window.c** defined in **xdg-shell-client-protocol.h**
- Assembles a command with the proxy marshall to the xdg_surface interface. With the ID of XDG_SURFACE_SET_FULLSCREEN 11
- The **xdg-shell-server-protocol.h** defines the **xdg_surface interface**, receives the message from the client: set_fullscreen!!
- In s**hell.c** look for the xdg_surface_implementation, its 11th method is:
- xdg_surface_set_fullscreen
- this method will get the output size (display size) and resize the window

SAMPLE CLIENT 3 WESTON-EVENTDEMO

Sample Client 3 Weston-EventDemo

The EventDemo will show us a great deal of the possible events ttat our clients can receive. Resize, focus, keys, buttons, axis and motion.

- Open the following files:
- CLIENT SIDE:
- eventdemo.c
- window.c
- shared/frame.c
- xdg-shell-client-protocol.h
- SERVER SIDE: Here you will look for the interface codes that will receive the messages that the client sends:
- xdg-shell-server-protocol.h difines the xdg_surface_interface.
- shell.c

Sample Client 3 Weston-EventDemo

The EventDemo will show us a great deal of the possible events ttat our clients can receive. Resize, focus, keys, buttons, axis and motion.

- Open the following files:
- CLIENT SIDE:
- eventdemo.c
- window.c
- shared/frame.c
- xdg-shell-client-protocol.h
- SERVER SIDE: Here you will look for the interface codes that will receive the messages that the client sends:
- xdg-shell-server-protocol.h difines the xdg_surface_interface.
- shell.c

Sample Client 3 Weston-EventDemo

- The first task, as always will be to connect to the main Wayland Display:
- //Step 1, 2 and 3: connects to the display and registers the client as a global event listener.
- d = display_create(&argc, argv);
-
- Afterwards, this client creates the window and surface.
- //Step 4, 5, and 6, creates the main client window
- e = eventdemo_create(d);
-         e->window = window_create(d);

Sample Client 3 Weston-EventDemo

- Within the same eventdemo_create function, the client will register its own callbacks to the handlers.
- //Step 7 Register the redraw_handler callback
- widget_set_redraw_handler(e->widget, redraw_handler);
- **the redraw_handler function will be drawn everytime a new frame needs to be redrawn.**
- Everytime the client requests a redraw with **window_schedule_redraw(e->window);**

Sample Client 3 Weston-EventDemo

- //Step 8 Register the resize_handler callback

- **widget_set_resize_handler(e->widget, resize_handler);**

- This function is called when the window needs to be resized. The window.c file functionality informs our window that it needs to be resized, by changing the widget width and height the window.c file functionality will take care of the rest.

- The resize function is triggered by a press and motion events on the FRAME borders.

Sample Client 3 Weston-EventDemo

- When a mouse button is pressed the pointer_handle_button is called.
- This will call the button_handler for the frame, which is:
- frame_button_handler → **window.c**
- if you press once the button you will call the function:
- frame_pointer_button → **shared/frame.c**
- frame_pointer_button_press → **shared/frame.c**
- Here the code will check WHERE did you press the button and set the frame status as FRAME_STATUS_RESIZE.

Sample Client 3 Weston-EventDemo

- Until the button is released, it will stay in the resize state, and every pointer movement will generate a resize event for the client.
- Now, if you move the pointer, the Server will send you the pointer_handle_motion event.
- pointer_handle_motion → window.c
- As you are still on the FRAME, the following function will be called:
- frame_motion_handler → will set the FRAME_STATUS_REPAINT to the flags and issue a repaint call.
- As the FRAME_STATUS_RESIZE is also set, the xdg surface resize function will be called.
- xdg_surface_resize → defined in xdg-shell-client-protocol.h, here a request to the server will be assembled with the XDG_SURFACE_RESIZE (6) index.
- The xdg_surface_implementation is defined in shell.c, look which function is in the 6th Index!
- xdg_surface_resize → shell.c, here you will send the new surface configuration to the compositor.

Sample Client 3 Weston-EventDemo

- //Step 9
- Register a callback to react when Wayland sends our application a message that we just got or lost focus.
- Once it receives the message, it will only print that it got focus or that it lost it.
- **window_set_keyboard_focus_handler(e->window,**
  keyboard_focus_handler);

- //Step 10
- Register a callback to react when Wayland sends our application a keyboard event, in this case we will only print the key that was received.
- **window_set_key_handler(e->window, key_handler);**

Sample Client 3 Weston-EventDemo

- //Step 11
- Register a callback to react when Wayland sends our application a button press within our client's surface.
- **widget_set_button_handler(e->widget, button_handler);**
-
- //Step 12
- Register a callback when Wayland sends a motion message, this message is only received when the mouse is moved WITHIN our application surface.
- **widget_set_motion_handler(e->widget, motion_handler);**
-
- //Step 13
- Register a callback when the mouse wheel is moved within our application.
- **widget_set_axis_handler(e->widget, axis_handler);**

SAMPLE CLIENT 4 WESTON-SIMPLE-TOUCH

Sample Client 4 Weston-Simple-Touch

- Simple-touch is a pretty interesting example, as it initializes by itself many of the Wayland structures and it does NOT use Cairo nor OpenGL ES 2.0 to render.
-
- For this client, open the files:
- CLIENT SIDE
- simple-touch.c
- wayland-client-protocol.h
- SERVER SIDE
- wayland-server-protocol.h
- wayland-shm.c

Sample Client 4 Weston-Simple-Touch

- Go to the main function, which calls the touch_create function, the first step, as always is getting the display connection:
-
- //Step 1 Create the display connection:
- touch->display = wl_display_connect(NULL);
-
- //Step 2 Get the display global registry
- touch->registry = wl_display_get_registry(touch->display);
-
- //Step 3 Add a listener to the global events that the registry will sent.
- wl_registry_add_listener(touch->registry, &registry_listener, touch);

Sample Client 4 Weston-Simple-Touch

- //Step 4 Dispatch the events to the display server, in this case we will dispatch our request to get the registry and register our global handler.
- wl_display_dispatch(touch->display);
-
- //Step 5 Wait for a display roundtrip, this ensures that our request gets to the server and that we receive all the events we registered for, in this case we will get info for EVERY INTERFACE on the Weston compositor.
- wl_display_roundtrip(touch->display);

Sample Client 4 Weston-Simple-Touch

- //Step 6 Create a compositor surface, this is a rectangular area that the compositor will display. This step does not specify the content nor buffers for the surface, only its main structure.
- touch->surface = wl_compositor_create_surface(touch->compositor);
-
- //Step 7 Assign the surface we created with a shell_surface role, it provides a way for our clients to interface with the surface so we can move it, resize it, assign priorities, etc.
- touch->shell_surface = wl_shell_get_shell_surface(touch->shell,
-   touch->surface);

Sample Client 4 Weston-Simple-Touch

- //Step 7.5 create a file within your os memory that will be passed to the wl_shm_create_pool argument. Using this file you will also be able to mmap it, so you can directly write to it using the CPU.
- fd = os_create_anonymous_file(size);
-         touch->data =
-                 mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
-
- //Step 8 Create a memory pool where a buffer can be created.
- pool = wl_shm_create_pool(touch->shm, fd, size);
- wl_shm_create_pool → wayland-client-protocol.h, assembles a message to the server with the index WL_SHM_CREATE_POOL (0), this will be sent to the Wayland server.
- Received by a wl_shm_interface in wayland-server-protocol.h
- It will execute the shm_create_pool in wayland-shm.c file, which will create the shm pool

Sample Client 4 Weston-Simple-Touch

- //Step 9 Create a buffer where you can render your actual data, A buffer will keep a reference to the pool it was created from so it is valid to destroy the pool immediately after creating a buffer from it.
- touch->buffer = wl_shm_pool_create_buffer(pool, 0,
-   touch->width, touch->height, stride,
-   WL_SHM_FORMAT_ARGB8888);
- wl_shm_pool_create_buffer → defined in wayland-client-protocol.h, assembles a message to the server with the ID WL_SHM_POOL_CREATE_BUFFER (0)
- wl_shm_pool_create_buffer → wayland-server-protocol.h, received by a wl_shm_pool_interface, the shm function corresponding to this id is: shm_pool_create_buffer defined in wayland-shm.c

Sample Client 4 Weston-Simple-Touch

- //Step 10 As the buffer keeps a reference to the pool, it is safe to destroy the pool handler.
- wl_shm_pool_destroy(pool);
-
- //Step 11 clear the area that we mapped on step 7.5. this is a great step, as it shows us, how to commit a surface to the compositor. As you can see, all writes to data, are actual writes to the buffer!!
- memset(touch->data, 64, width * height * 4);
- wl_surface_attach(touch->surface, touch->buffer, 0, 0);
- wl_surface_damage(touch->surface, 0, 0, width, height);
- wl_surface_commit(touch->surface);

Sample Client 4 Weston-Simple-Touch

- //Step 10 As the buffer keeps a reference to the pool, it is safe to destroy the pool handler.
- wl_shm_pool_destroy(pool);
-
- //Step 11 clear the area that we mapped on step 7.5. this is a great step, as it shows us, how to commit a surface to the compositor. As you can see, all writes to data, are actual writes to the buffer!!
- memset(touch->data, 64, width * height * 4);
- wl_surface_attach(touch->surface, touch->buffer, 0, 0);
- wl_surface_damage(touch->surface, 0, 0, width, height);
- wl_surface_commit(touch->surface);

Sample Client 4 Weston-Simple-Touch

- //Step 12 as the global_handler installed listener for the seat inputs, you will now have the chance to paint whenever the touch screen is stimulated.
- In order to do so you only need to modify the touch->data and attach and commit the surface.
- wl_surface_attach(touch->surface, touch->buffer, 0, 0);
-         wl_surface_damage(touch->surface, x - 2, y - 2, 5, 5);
-
- //Step 13
- As you will only render when you receive an event, your main loop can only be a display_dispatch function.
- That will automatically execute the handlers for the listeners that you registered.
- wl_surface_attach(touch->surface, touch->buffer, 0, 0);
-         ret = wl_display_dispatch(touch->display);

CREATING YOUR OWN WAYLAND BASED APPLICATION

Creating Your Own Wayland Application

- We now have enough knowledge regarding how clients work. So we are able to create our OWN.
- We are not going to use the window.c file, as it includes too much complexity, our application will rely on a single file.
-
- We want it to be able to :
- Read Inputs from Keyboard, mouse and touch.
- Modify window size.
- Send a full screen event.
- Render Content using OpenGL ES 2.0 Content

Creating Your Own Wayland Application

- Open the Simple.c file that I provided.
- You will find all the familiar steps that we covered on the previous code review.
-
- The first step will be to connect to the main display.
- //STEP 1
- sdisplay.display = wl_display_connect(NULL);
-
- The second step will creates a registry object that allows us to list the global objects from the compositor
- //STEP 2
- sdisplay.registry = wl_display_get_registry(sdisplay.display);
-
- Next we add a listener to the global objects that the compositor throws at us.
- //STEP3
- wl_registry_add_listener(sdisplay.registry,
-   &registry_listener, &sdisplay);

Creating Your Own Wayland Application

- Next we need to dispatch all the above request to the server.
- //STEP 4
- wl_display_dispatch(sdisplay.display);
-
- After that the compositor will start sending events to our client, and as we registered as listeners, we will execute the global handler everytime the compositor sends us a global object. This function ensures that all the pending requests and events are processed befor continuing.
- During this step, we will also register our seat listener, that will enable us listening to a Mouse, Keyboard and Pointer.
- //STEP 5
- wl_display_roundtrip(sdisplay.display);

Creating Your Own Wayland Application

- The next step is to create the compositor surface, it only defines a region on the compositor that will be rendered to the final display, but it is only a data structure, it does not include a buffer.
- //STEP 6
- swindow.surface = wl_compositor_create_surface(sdisplay.compositor);

- Once you have the surface you need to provide it a role, as we want to resize, move and have it as fullscreen, it will be a shell_surface.
- //STEP 7
- swindow.shell_surface = wl_shell_get_shell_surface(sdisplay.shell,
-   swindow.surface);

Creating Your Own Wayland Application

- Then you need it to take a place in the view hierarchy of the Weston Compositor, that is why we set it as top:
- //STEP 8
- wl_shell_surface_set_toplevel(swindow.shell_surface);
-
- After this is only a matter of initializing the EGL Display using the Wayland Display as the native display.
- //STEP 9
- eglNativeDisplayType = sdisplay.display;
- egldisplay = eglGetDisplay(eglNativeDisplayType);

Creating Your Own Wayland Application

- Then you need to create the EGL Window from the Wayland surface we created on Step 6
- //STEP 10
- swindow.native =
- wl_egl_window_create(swindow.surface, swindow.window_size.width,
-   swindow.window_size.height);
-
- And finally you create the buffer where your GPU will render, this buffer is the one that the compositor will receive when composing the display.
- //STEP 11
- eglsurface = eglCreateWindowSurface(egldisplay, eglconfig, eglNativeWindow, NULL);

ADVANCED SETUPS: EXTENDED DESKTOP

Extended Desktop

- Extended desktop is an interesting approach, as 2 screen framebuffers will be combined into a single one that will be shown across multiple displays.

- This functionality is only supported using the GAL2D blitter, in order to enable a multiple desktop approach, you need to pass the following parameters to your weston command:
- /etc/init.d/weston stop
- echo 0 > /sys/class/graphics/fb4/blank
- weston --tty=1 --use-gal2d=1 --use-gl=0 --device=/dev/fb0,/dev/fb4 &

- With this, 2 contiguous surfaces will be created, and the gal2d compositor will output the 2 main buffers to their respective display. And when a surface is shared between the 2 displays, the compositor will divide it so a fragment of each screen is shown on each buffer.

ADVANCED SETUPS: MULTIPLE SEATS

Multiple Seats

- A seat is a group of input devices, like keyboards, mice and touch devices:
- The object wl_seat is published as a global during start up (via the registry) or when an input device is hot plugged. A seat has a pointer and mantains a keyboard focus and a pointer focus. When we register a seat listener, we need to define its capabilities
- The seat_handle_capabilities (we named it that way on Simple.c) function will be called once per each seat.
- Our current weston implementation requires a slight change in order to handle multiple seats.
- You will need to modify the compositor-fbdev.c and launcher-util.c files, both of those files are hard coded to always use the same seat.
- In our case we will remove that hard coding and ensure that each weston instance that is launched has a different seat.

Multiple Seats

- mkdir /run/test0
- mkdir /run/test1
- echo 0 > /sys/class/graphics/fb4/blank
- export FB_MULTI_BUFFER=2
- export XDG_RUNTIME_DIR=/run/test0
- weston --seat=FTF01 --tty=1 --log=/var/log/weston0.log --use-gal2d=1 --backend=fbdev-backend.so --device=/dev/fb0 -i0 &
- weston-simple-egl &
- export XDG_RUNTIME_DIR=/run/test1weston --seat=FTF02 --tty=1 --log=/var/log/weston1.log --use-gal2d=1 --backend=fbdev-backend.so --device=/dev/fb4 -i0 &
- weston-simple-egl &
- With this you will have an independent display with an independent set of inputs in your system.

ADVANCED SETUPS: MULTIPLE XWAYLAND

X Wayland

- Wayland is a complete window system in itself, but even so, if we're migrating away from X, it makes sense to have a good backwards compatibility story. With a few changes, the Xorg server can be modified to use wayland input devices for input and forward either the root window or individual top-level windows as wayland surfaces.

- The server still runs the same 2D driver with the same acceleration code as it does when it runs natively. The main difference is that Wayland handles presentation of the windows instead of KMS.

X Wayland

- In order to add Xwayland support for our BSPs you need to download yocto as explained in previous slides.

- You will create a new DISTRO using:
- DISTRO=fsl-imx-xwayland MACHINE=imx6qsabresd source ./fsl-setup-release.sh -b build-xwayland

- Then bitbake fsl-image-guiOnce you have the image your Wayland/Weston image will be able to run X11 applications seamlessly!!

- Excepting X11 applications that use EGL, we don’t support that, if you plan to use EGL apps, please use the Wayland provided functions to create the buffer.

PERFORMANCE COMPARISON AGAINST X11 ON NXP DEVICES

X11 Rendering a 1080p EGL – GLES 3 application

- MMDC with a simple triangle application running
- MMDC new Profiling results:
- ***********************
- Measure time: 500ms
- Total cycles count: 264079768
- Busy cycles count: 240126027
- Read accesses count: 10510884
- Write accesses count: 12139903
- Read bytes count: 652087272
- Write bytes count: 771126472
- Avg. Read burst size: 62
- Avg. Write burst size: 63
- Read: 1243.76 MB/s / Write: 1470.81 MB/s Total: 2714.56 MB/s
- Utilization: 37%
- Overall Bus Load: 90%
- Bytes Access: 62

Wayland Rendering a 1080p EGL – GLES 3 application

- MMDC new Profiling results:
- ***********************
- Measure time: 501ms
- Total cycles count: 264086968
- Busy cycles count: 175151354
- Read accesses count: 10144118
- Write accesses count: 5884949
- Read bytes count: 633201560
- Write bytes count: 375039272
- Avg. Read burst size: 62
- Avg. Write burst size: 63
- Read: 1205.33 MB/s / Write: 713.90 MB/s Total: 1919.23 MB/s
- Utilization: 35%
- Overall Bus Load: 66%
- Bytes Access: 62

SECURE CONNECTIONS
FOR A SMARTER WORLD

ATTRIBUTION STATEMENT