A Current Overview of the DRM KMS Driver-Side APIs

Paul Kociatkowski
paul@bootlin.com
Paul Kocikowskki

- Embedded Linux engineer at Bootlin
  - Embedded Linux expertise
  - Development, consulting and training
  - Strong open-source focus

- Open-source contributor
  - Contributor to the cedrus Allwinner Video Engine V4L2 driver
  - Contributor to the sun6i-csi Allwinner video capture V4L2 driver
  - Author of the sun6i-isp and sun6i-mipi-csi2 Allwinner video capture V4L2 drivers
  - Author of the ov5648 and ov8865 camera sensor V4L2 drivers
  - Contributor to the sun4i-drm and vc4 DRM drivers
  - Author of the logicvc-drm DRM driver
  - Author of the displaying and rendering graphics with Linux training

- Living in Toulouse, south-west of France
Introduction
Display Hardware Pipeline

- **Framebuffer**: Memory buffer(s) with pixel data
- **Plane**: Pixel mixing (rotation, scaling, format and more), layers
- **CRTC**: Timings generation and pixel streaming
- **Encoder**: Interface physical adaptation (protocol, signals)
- **Bridge**: Interface transcoding
- **Panel**: Display surface emitting/reflecting light
- **Monitor**: Peripheral integrating a panel
Display Hardware Overview

Typical hardware components:
- Display controller in (PCIe) graphics card (x86) or as unit in SoC (embedded)
- On-board and internal bridge(s)
- Display connector(s), always-connected panel(s)

Memory cases:
- Dedicated memory in graphics card (video RAM)
- Shared memory with the system
  - With IOMMU: general system pages, scatter-gather
  - Without IOMMU: reserved contiguous memory
- CPU access and cache management:
  - Automatic with cache-coherent interconnect
  - Manual cache invalidate/flush or cache disabled otherwise
Linux userspace APIs for display:

▶ **fbdev** interface: deprecated
  - Simple uAPI with pre-allocated memory
  - Very limited use-cases and configuration
  - Bad performance (no zero-copy)
  - No new drivers, will be removed eventually

▶ **DRM KMS/DRM mode** uAPI:
  - Proper pipeline configuration (planes, CRTC, connector)
  - Well-balanced abstraction of hardware complexity
  - DRM Atomic uAPI
  - DRM GEM/TTM memory management
  - DRM Prime for dma-buf zero-copy
  - DRM Sync Object for fences
  - Various drivers for graphics cards and embedded
Userspace software status:

- DRM KMS supported and used by most components
- fbdev still supported as fallback
- fbdev still used by quick-and-dirty projects (please stop)

Userspace software display support:

- **Display servers**: weston, sway, mutter, kwin, xorg
- **Graphics libraries**: SDL, Qt (eglfs), DirectFB2, LVGL, Mesa (GBM)
- **Media libraries**: GStreamer (kmssink), FFmpeg (kmsgrab), libcamera
- **Tools**: modetest, igt, kmscube, glmark2
DRM KMS Internals
DRM KMS Components Overview

Kernel hardware components:

▶ Drivers registered from bus infrastructure:
  • pci bus for graphics cards
  • platform bus for SoC units
  • i2c, spi, etc for bridges and panels
  • platform or mipi_dsi for panels

▶ Drivers register components to DRM KMS framework:
  • struct drm_device, for display controllers
  • struct drm_bridge, for bridges (internal or external)
  • struct drm_panel, for panels

▶ Drivers need to identify each other:
  • Represent pipeline topology
  • Retrieve remote component structures for API use
  • Static description via device-tree graph with port/endpoint (or ACPI)
  • Node nesting for bus (DSI)
tcon0: lcd-controller@1c0c000 {
    compatible = "allwinner,sun50i-a64-tcon-lcd",
                 "allwinner,sun8i-a83t-tcon-lcd";
    [...] 
}

ports {
    [...] 
}

tcon0_out: port@1 {
    reg = <1>;
    [...] 
}

tcon0_out_dsi: endpoint@1 {
    reg = <1>;
    remote-endpoint = <&dsi_in_tcon0>;
    allwinner,tcon-channel = <1>;
    [...] 
}

dsi: dsi@1ca0000 {
    compatible = "allwinner,sun50i-a64-mipi-dsi";
    [...] 
}

    port {
        dsi_in_tcon0: endpoint {
            remote-endpoint = <&tcon0_out_dsi>;
        };
    };

    panel@0 {
        compatible = "xingbangda,xbd599";
        reg = <0>;
        [...] 
    };
};
DRM Display Controller: Base Driver

- Driver data static declaration: `struct drm_driver`,
  - `driver_features`: bitfield of `DRIVER_MODESET`, `DRIVER_ATOMIC`, `DRIVER_GEM`
  - `fops`: default definitions with `DEFINE_DRM_GEM_FOPS`
  - `name`, `desc`, `date/major/minor` for information
  - Various operation callbacks, default definitions with `DRM_GEM_DMA_DRIVER_OPS`

- Device data: `struct drm_device`
  - Created by the DRM framework
  - Associated with character devices (card, render) for uAPI

- Setup at `probe()`:
  - Allocate `struct drm_device` from `struct drm_driver` with `drm_dev_alloc()`
    devm-managed with `devm_drm_dev_alloc()`, can allocate parent structure
  - Check if mode setting is allowed with `drm_firmware_drivers_only()`
    honor the `nomodeset` kernel command line parameter
  - Initialize display controller components
  - Register device with `drm_dev_register()`
Components registration typical order:
- Planes with `struct drm_plane`
- CRTC with `struct drm_crtc`, attach to planes
- Encoder with `struct drm_encoder`, attach to CRTC
- Connector with `struct drm_connector`, attach to encoder

Cleanup at `remove()`:
- Unregister with `drm_dev_unregister()`
- Call shutdown helper `drm_atomic_helper_shutdown()` to stop CRTCs
- Put reference (non-devm) with `drm_dev_put()`

Power management at `suspend()`/`resume()`:
- Pipeline configuration saved/disabled at suspend with:
  `drm_mode_config_helper_suspend()`
- Pipeline configuration restored/enabled at resume with:
  `drm_mode_config_helper_resume()`
Translation Table Manager (TTM) memory manager:

- Historic memory manager for DRM, big and complex
- Supports both shared system memory and dedicated video memory, used by graphics card drivers

Graphics Execution Manager (GEM) memory manager:

- New design from Intel, focused on sharing code
- Only supports shared system memory, used by most embedded drivers
- Provides `struct drm_driver` fops and callbacks with `DEFINE_DRM_GEM_FOPS` and `DRM_GEM_DMA_DRIVER_OPS`
- Supports driver-specific `dumb_create` operation with `DRM_GEM_DMA_DRIVER_OPS_WITH_DUMB_CREATE`
- Allocates write-combined DMA buffers with `dma_alloc_wc()` contiguous or not depending on IOMMU presence, coherent
- Also supports non-coherent (requires explicit sync)
- Helper to get framebuffer DMA address: `drm_fb_dma_get_gem_addr()`
GEM can typically use the Contiguous Memory Allocator (CMA):
- Meant for large contiguous buffer allocation
- Uses reclaimable reserved pools of (DRAM) memory
- Reserved early at boot with static size

Default system pool available to every device:
- Size configured with the `CONFIG_CMA_SIZE_MBYTES` option or with the `cma` kernel command line parameter

Dedicated pool attached to specific devices:
- Declared in device-tree with compatible `shared-dma-pool` under `reserved-memory` node
- Attached to node in device-tree with `memory-region` property
- Attached to `struct device` with `of_reserved_mem_device_init()`
- Detached from `struct device` with `of_reserved_mem_device_release()`
- DMA allocations with `struct device` then automatically use dedicated pool
- Can be made default pool with `linux,cma-default;` in device-tree
/ {
    [...]  

    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;

        gfx_memory: framebuffer {
            size = <0x01000000>;
            alignment = <0x01000000>;
            compatible = "shared-dma-pool";
            reusable;
        }
    };
};

gfx: display@1e6e6000 {
    compatible = "aspeed,ast2600-gfx", "syscon";
    [...]  

    memory-region = <&gfx_memory>;
};
DRM Display Controller: Mode Config

- **Mode config data:** `struct drm_mode_config` from `struct drm_device`
  - `{min, max}_{width, height}`: framebuffer dimension limits
  - `preferred_depth`: default framebuffer pixel depth
  - `funcs`: driver-specific `struct drm_mode_config_funcs`

- **Mode config functions:** `struct drm_mode_config_funcs` (boilerplate)
  - `fb_create`: framebuffer creation with `drm_gem_fb_create()`
  - `atomic_check`: atomic commit validation with `drm_atomic_helper_check()`
  - `atomic_commit`: atomic commit entry point with `drm_atomic_helper_commit()`

- **Setup at `probe()`:**
  - Initialize mode config with `drm_mode_config_init()` automatically calls `drm_mode_config_cleanup()` using destroy functions
  - Configure mode config data fields
  - Reset global pipeline state with `drm_mode_config_reset()` using reset functions
  - Register device with `drm_dev_register()`
Atomic support:
- Group batches of changes together as atomic commit
- Provided by userspace as a list of property changes
- Atomic state managed by the KMS framework

Atomic state: `struct drm_atomic_state`
- New/old configuration state for internal components
- Derived to component-specific structures
- Used to configure hardware registers

Non-atomic is considered legacy and not covered here
- Expected that new KMS drivers are atomic nowadays
- Converting non-atomic drivers is welcome
DRM Display Controller: Planes

▶ Plane data: **struct drm_plane**
  • type: one of DRM_PLANE_TYPE_PRIMARY, DRM_PLANE_TYPE_OVERLAY, DRM_PLANE_TYPE_CURSOR
  • possible_crtcs: valid CRTCs with drm_crtc_mask()
  • formats: list of supported pixel formats
  • modifiers: list of supported pixel format modifiers (tiling, etc)

▶ Plane functions: **struct drm_plane_funcs** (boilerplate)
  • reset: plane state reset with drm_atomic_helper_plane_reset()
  • destroy: plane destruction with drm_plane_cleanup()
  • atomic_{duplicate,destroy}_state: plane state handling with
    drm_atomic_helper_plane_duplicate_state(),
    drm_atomic_helper_plane_destroy_state()
  • {update,disable}_plane: plane configuration with
    drm_atomic_helper_update_plane(), drm_atomic_helper_disable_plane()
DRM Display Controller: Planes

▶ Plane atomic state: `struct drm_plane_state`
  - New/old plane states available from `struct drm_atomic_state`:
    `drm_atomic_get_new_plane_state()`, `drm_atomic_get_old_plane_state()`
  - Associated CRTC: `struct drm_crtc`
  - Associated framebuffer: `struct drm_framebuffer`
  - Various base and optional properties

▶ Plane helper functions: `struct drm_plane_helper_funcs`
  - `atomic_check`: driver-specific atomic state validation
    can use `drm_atomic_helper_check_plane_state()` with scaling/position features
  - `atomic_update`: driver-specific plane configuration
    register configuration using plane (and crtc) atomic state
  - `atomic_disable`: driver-specific plane disable

▶ Setup at `probe()`:
  - Initialize and register `struct drm_plane` with `drm_universal_plane_init()`
  - Register plane helpers with `drm_plane_helper_add()`
  - Configure available plane properties
Plane properties are exposed to userspace and used in configuration.

Base properties are registered with `drm_universal_plane_init()`.

Optional properties can be registered by driver:

- Plane-wide alpha: `drm_plane_create_alpha_property()`
- Plane stacking order: `drm_plane_create_zpos_property()`, `drm_plane_create_zpos_immutable_property()`
- Rotation: `drm_plane_create_rotation_property()`
- Blend mode: `drm_plane_create_blend_mode_property()`
- Scaling filter: `drm_plane_create_scaling_filter_property()`
- Custom driver-specific properties may also exist.
DRM Display Controller: Metadata

- **Display mode:** `struct drm_display_mode`
  - Describes display timings and *some* signal characteristics (sync polarity, composite sync, double/half clock rate)
  - List provided by connector, with preferred indication
  - Chosen by userspace to configure CRTC with property

- **Display information:** `struct drm_display_info`
  - Describes pixel interface characteristics
  - `bus_formats`: list of `MEDIA_BUS_FMT_*` formats
  - `bus_flags`: bitfield of `DRM_BUS_FLAG_*` for signal characteristics
  - `{width, height}_mm`: physical surface dimensions

- Both are retrieved either:
  - Dynamically with EDID (`struct edid`) via DDC for monitors
  - Statically with hardcoded definitions for panels
DRM Display Controller: Connector

▶ Connector data: `struct drm_connector`
- **type**: display interface indication, `DRM_MODE_CONNECTOR_*`
- **status**: one of `connector_status_connected`, `connector_status_disconnected`, `connector_status_unknown`
- **possible_encoders**: valid encoders with `drm_encoder_mask()`
- **modes**: list of available display interface modes

▶ Connector functions: `struct drm_connector_funcs` (boilerplate)
- **reset**: connector state reset with `drm_atomic_helper_connector_reset()`
- **destroy** connector destruction with `drm_connector_cleanup()`
- **atomic_{duplicate,destroy}_state**: connector state handling with `drm_atomic_helper_connector_duplicate_state()`, `drm_atomic_helper_connector_destroy_state()`
- **fill_modes**: get mode list from available sources with `drm_helper_probe_single_connector_modes()`
 DRM Display Controller: Connector

- **Connector atomic state:** `struct drm_connector_state`
  - New/old connector states available from `struct drm_atomic_state`:
    - `drm_atomic_get_new_connector_state()`,
    - `drm_atomic_get_old_connector_state()`
  - Associated CRTC and encoder: `struct drm_crtc`, `struct drm_encoder`
  - Various base and optional properties

- **Connector helper functions:** `struct drm_connector_helper_funcs`
  - `get_modes`: retrieve list of modes with `drm_get_edid()` and
    - `drm_add_edid_modes()` or `drm_panel_get_modes()`
  - `mode_{valid,fixup}`: validate/fixup proposed mode with hardware constraints
  - `detect`: detect connector status

- **Setup at `probe()`:**
  - Initialize and register `struct drm_connector` with `drm_connector_init()`
  - Register connector helpers with `drm_connector_helper_add()`
  - Attach encoder to connector with `drm_connector_attach_encoder()`
DRM Display Controller: Connector Hotplug

- Connector status detection:
  - Detect connector plug/unplug for monitors
  - Using dedicated pin or status, associated interrupt or not
  - Updates `struct drm_connector` status using `detect` helper function
  - Notify userspace via sysfs uevent `HOTPLUG=1` (and `CONNECTOR=*`)

- Interrupt-based detection:
  - Set `struct drm_connector` polled field to `DRM_CONNECTOR_POLL_HPD`
  - Event reported from IRQ handler with `drm_connector_helper_hpd_irq_event()` or `drm_helper_hpd_irq_event()` (global)

- Active polling (10 Hz):
  - Set `struct drm_connector` polled field to `DRM_CONNECTOR_POLL_CONNECT | DRM_CONNECTOR_POLL_DISCONNECT`
  - Dedicated worker started with `drm_kms_helper_poll_init()`
  - Dedicated worker stopped with `drm_kms_helper_poll_fini()`
DRM Display Controller: CRTC

- CRTC data: `struct drm_crtc`
  - `primary, cursor`: legacy planes for compatibility (replaced by atomic state)
  - `mode`: legacy mode for compatibility (replaced by atomic state)

- CRTC functions: `struct drm_crtc_funcs` (mostly boilerplate)
  - `reset`: crtc state reset with `drm_atomic_helper_crtc_reset()`
  - `destroy`: crtc destruction with `drm_crtc_cleanup()`
  - `atomic_{duplicate,destroy}_state`: crtc state handling with
    `drm_atomic_helper_crtc_duplicate_state()`,
    `drm_atomic_helper_crtc_destroy_state()`
  - `set_config`: crtc configuration with `drm_atomic_helper_set_config()`
  - `page_flip`: crtc page flip with `drm_atomic_helper_page_flip()`
  - `enable_vblank`: driver-specific vblank interrupt enable
  - `disable_vblank`: driver-specific vblank interrupt disable
CRTC atomic state: \textit{struct drm_crtc_state}

- New/old crtc states available from \textit{struct drm_atomic_state}:
  
  \texttt{drm_atomic_get_new_crtc_state()}, \texttt{drm_atomic_get_old_crtc_state()}

- Associated planes, connectors and encoders with \texttt{drm_plane_mask()},
  \texttt{drm_connector_mask()}, \texttt{drm_encoder_mask()}

- Adjusted and requested modes: \texttt{adjusted_mode}, \texttt{mode}

- Pending vblank event with \textit{struct drm_pending_vblank_event}

- Various properties (gamma, scaling filter)

CRTC helper functions: \textit{struct drm_crtc_helper_funcs}

- \texttt{mode\_\{valid,fixup\}}: validate/fixup proposed mode with hardware constraints
- \texttt{atomic\_check}: driver-specific atomic state validation
- \texttt{atomic\_enable}: driver-specific crtc configuration using atomic state,
  enable vblank with \texttt{drm_crtc_vblank_on()}
- \texttt{atomic\_disable}: driver-specific crtc disable,
  disable vblank with \texttt{drm_crtc_vblank_off()}
DRM Display Controller: CRTC

Setup at `probe()`:
- Initialize and register `struct drm_crtc` with `drm_crtc_init_with_planes()`
- Register CRTC helpers with `drm_crtc_helper_add()`
- Provide port `struct device_node` with `of_graph_get_port_by_id()`, used by `drm_of_find_possible_crtcs()` with multiple device-tree nodes

Vblank reporting with `drm_crtc_handle_vblank()` in interrupt handler:
- Wake-up any task waiting for vblank

Vblank userspace event handling: `struct drm_pending_vblank_event`
- Locked with `struct drm_device` event_lock
- Grabbed at `atomic_enable` from atomic state event with `drm_crtc_vblank_get()` copied and removed from `struct drm_crtc_state`
- Returned in interrupt handler with `drm_crtc_send_vblank_event()` and `drm_crtc_vblank_put()`
DRM Display Controller: Encoder

- **Encoder data:** `struct drm_encoder`
  - `type`: physical encoding indication, `DRM_MODE_ENCODER_*`
  - `possible_crtcs`: valid CRTCs with `drm_crtc_mask()`

- **Encoder functions:** `struct drm_encoder_funcs` (mostly boilerplate)
  - `reset`: encoder state reset (optional)
  - `destroy`: encoder destruction with `drm_encoder_cleanup()`
No encoder atomic state, using crtc and connector state

Encoder helper functions: `struct drm_encoder_helper_funcs`

- `mode_{valid,fixup}`: validate/fixup proposed mode with hardware constraints
- `atomic_check`: driver-specific atomic state validation
- `atomic_enable`: driver-specific encoder configuration using atomic state
- `atomic_disable`: driver-specific encoder disable

Setup at `probe()`:

- Initialize and register `struct drm_encoder` with `drm_encoder_init()` or `drm_simple_encoder_init()` (boilerplate funcs)
- Register encoder helpers with `drm_encoder_helper_add()`
- Attach encoder to connector with `drm_connector_attach_encoder()`
DRM Bridge

- **Bridge data:** `struct drm_bridge`
  - `ops`: bitfield of `DRM_BRIDGE_OP_DETECT`, `DRM_BRIDGE_OP_EDID`, `DRM_BRIDGE_OP_HPD`, `DRM_BRIDGE_OP_MODES`
  - `type`: terminal connector type, `DRM_MODE_CONNECTOR_*`
  - `timings`: optional `struct drm_bridge_timings` with input bus flags
  - `chain_node`: list of `struct drm_bridge` for chaining bridges
  - `encoder`: `struct drm_encoder` currently attached to the bridge
  - Not tied to a specific `struct drm_device`

- **Bridge functions:** `struct drm_bridge_funcs` (boilerplate)
  - `atomic_{duplicate,destroy}_state`: bridge state handling with `drm_atomic_helper_bridge_duplicate_state()`, `drm_atomic_helper_bridge_destroy_state()`.
  - `atomic_reset`: bridge state reset with `drm_atomic_helper_bridge_reset()`.
  - No cleanup callback since `drm_mode_config_cleanup()` relates to `struct drm_device`
DRM Bridge

- Bridge state: `struct drm_bridge_state` (useful for chaining)
  - New/old bridge states available from `struct drm_atomic_state`:
    - `drm_atomic_get_new_bridge_state()`, `drm_atomic_get_old_bridge_state()`
  - Input/output bus configuration: `struct drm_bus_cfg` with
    `MEDIA_BUS_FMT_*` if available or `MEDIA_BUS_FMT_FIXED`

- Bridge functions: `struct drm_bridge_funcs`
  - `attach/detach`: create/destroy connector, unless
    `DRM_BRIDGE_ATTACH_NO_CONNECTOR` flag is specified
    or attach/detach next bridge with `drm_bridge_attach()`
  - `mode_{valid,fixup}`: validate/fixup proposed mode with hardware constraints
  - `atomic_get_{input,output}_bus_fms`: report supported input/output bus formats for negotiation in bridge chains
  - `atomic_enable`: driver-specific bridge configuration using (crtc) atomic state
  - `atomic_disable`: driver-specific bridge disable
  - `atomic_check`: driver-specific atomic state validation
Setup at `probe()` (dedicated driver):

- Configure `type`, `ops`, `funcs`, `of_node` and possibly timings
- Initialize and register `struct drm_bridge` with `drm_bridge_add()`
  devm-managed with `devm_drm_bridge_add()`

Cleanup at `remove()`:

- Unregister and cleanup with `drm_bridge_remove()`

Display controller driver usage:

- Identified via device-tree with `drm_of_find_panel_or_bridge()` or `devm_drm_of_get_bridge()`
- Attached to encoder with `drm_bridge_attach()`,
- Automatically detached during encoder cleanup
- Connector is created by (final) bridge directly
- Bridge functions are called automatically once attached with encoder
DRM Panel

- **Panel data:** `struct drm_panel`
  - `backlight`: `struct backlight_device` for the attached backlight
  - `connector_type`: display interface indication, `DRM_MODE_CONNECTOR_*`
  - Not tied to a specific `struct drm_device`

- **Panel functions:** `struct drm_panel_funcs`
  - `prepare`: Setup panel (typically power lines, configuration)
  - `enable`: Turn on panel, start expecting data
  - `disable`: Turn off panel, stop expecting data
  - `unprepare`: Cleanup panel
  - `get_{modes,timings}`: return list of supported modes/timings

- **No panel atomic state:**
  - Timings are known in advance with a single mode (typical)
  - Generally no need to configure timings explicitly
DRM Panel

- **Setup at** `probe()` *(dedicated driver):*
  - Initialize panel with `drm_panel_init()` given `funcs` and `type`
  - Attach backlight via device-tree using `drm_panel_of_backlight()`
  - Register panel with `drm_panel_add()`

- **Cleanup at** `remove()`:
  - Unregister panel with `drm_panel_remove()`

- **Display controller driver usage (deprecated):**
  - Identified via device-tree with `drm_of_find_panel_or_bridge()`
  - Register encoder and connector for panel
  - Return panel modes in connector `get_modes` with `drm_panel_get_modes()`
  - Enable panel in encoder `atomic_enable` with `drm_panel_prepare()` and `drm_panel_enable()` *(automatically enables backlight)*
  - Disable panel in encoder `atomic_disable` with `drm_panel_unprepare()` and `drm_panel_disable()`
Differentiated handling for bridges and panels is painful:

- Both are *in-fine* connected to an encoder
- Panels require drm device to manage the connector
- Bridges register their own connector
- Provided functions are comparable (enable, disable, modes list)

DRM Panel Bridge API closes the gap:

- Unified API for both, using `struct drm_bridge`
- Boilerplate connector registered transparently for panels
- Calls back panel functions in bridge functions

Display controller driver usage:

- Use `devm_drm_of_get_bridge()` instead of `drm_of_find_panel_or_bridge()`
- Use bridge normally, no particular difference
DRM Bridge and Panel Drivers

- Generic drivers provide static data and regulator/gpio integration

- Generic bridge drivers:
  - `simple-bridge`: static bridge timings with dedicated device-tree compatibles
  - `lvds-codec`: generic device-tree compatibles
  - `display-connector`: generic device-tree compatibles

- Generic panel drivers:
  - `panel-simple`: static modes list, panel-specific device-tree compatibles
  - `panel-lvds`: timings from device-tree properties
  - `panel-edp`: static modes list, panel-specific device-tree compatibles, edid fixup tables

- Specific drivers:
  - Generally require specific register configuration
  - Pitfall: panel and LCD controller confusion
  - Device-tree compatible must **not** be the LCD controller name
  - Having a common driver for a LCD controller is a good idea (with panel-specific compatibles)
static const struct drm_display_mode
lemaker_bl035_rgb_002_mode = {
    .clock = 7000,
    .hdisplay = 320,
    .hsync_start = 320 + 20,
    .hsync_end = 320 + 20 + 30,
    .htotal = 320 + 20 + 30 + 38,
    .vdisplay = 240,
    .vsync_start = 240 + 4,
    .vsync_end = 240 + 4 + 3,
    .vtotal = 240 + 4 + 3 + 15,
};

static const struct of_device_id platform_of_match[] = {
    [...]  
    {  
        .compatible = "lemaker,bl035-rgb-002",
        .data = &lemaker_bl035_rgb_002,
    },  
    [...]  
};

static const struct panel_desc
lemaker_bl035_rgb_002 = {
    .modes = &lemaker_bl035_rgb_002_mode,
    .num_modes = 1,
    .size = {
        .width = 70,
        .height = 52,
    },
    .bus_format = MEDIA_BUS_FMT_RGB888_1X24,
    .bus_flags = DRM_BUS_FLAG_DE_LOW,
};
DRM Repositories and Lists

Repositories:

▶ DRM (top): https://cgit.freedesktop.org/drm/drm
  • Branches: drm-next, drm-fixes
  • Maintainer: Dave Airlie

▶ DRM Misc: https://cgit.freedesktop.org/drm/drm-misc
  • Branches: drm-misc-next, drm-misc-fixes
  • Maintainers: Maarten Lankhorst, Maxime Ripard, and Thomas Zimmermann

▶ Hardware-specific:
  • DRM Intel: https://cgit.freedesktop.org/drm/drm-intel
  • DRM AMD: https://gitlab.freedesktop.org/agd5f/linux.git
  • DRM Nouveau: https://gitlab.freedesktop.org/drm/nouveau

Patch submission and tracking:

▶ Patchwork: https://patchwork.freedesktop.org
▶ Mailing list: dri-devel@lists.freedesktop.org
Questions? Suggestions? Comments?

Paul Kociatkowski
paul@bootlin.com

Slides under CC-BY-SA 3.0
https://bootlin.com/pub/conferences/