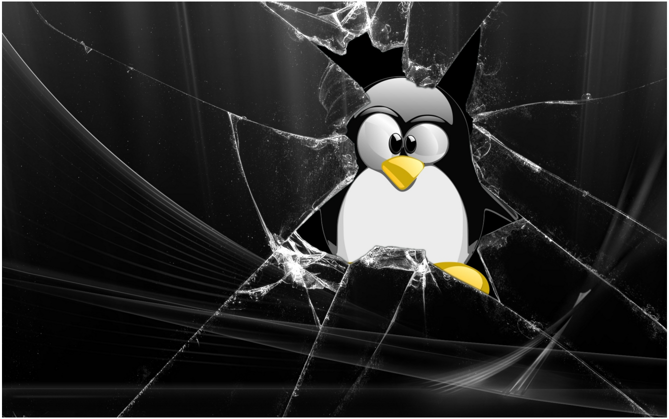
The DRM/KMS subsystem from a newbie's point of view



Boris Brezillon

- Embedded Linux engineer and trainer at Bootlin
  - Embedded Linux and Android **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux, Linux driver development, Android system and Yocto/OpenEmbedded **training courses**, with materials freely available under a Creative Commons license.
  - `http://bootlin.com`
- Contributions
  - **Kernel support for the AT91 SoCs** ARM SoCs from Atmel
  - **Kernel support for the sunXi SoCs** ARM SoCs from Allwinner
- Living in **Toulouse**, south west of France

Agenda

Context: What is this talk about?

- Sharing my understanding of the DRM/KMS subsytem learned while working on the Atmel HLCDC driver
- Explaining some key aspects (from my point of view) of the DRM/KMS subsystem
- Explaining some common concepts in the video/graphic world and showing how they are implemented in DRM/KMS
- Sharing some tips on how to develop a KMS driver based on my experience
- This talk is not:
  - A detailed description of the DRM/KMS subsystem
  - A description on how to use a DRM device (user-space API)
  - And most importantly: this talk is not given by an expert
- Don't hesitate to correct me if you think I'm wrong ;-)

Context: How to display things in the Linux world

- Different solutions, provided by different subsystems:
  - FBDEV: Framebuffer Device
  - DRM/KMS: Direct Rendering Manager / Kernel Mode Setting
  - V4L2: Video For Linux 2
- How to choose one: it depends on your needs
  - Each subsytem provides its own set of features
  - Different levels of complexity
  - Different levels of activity

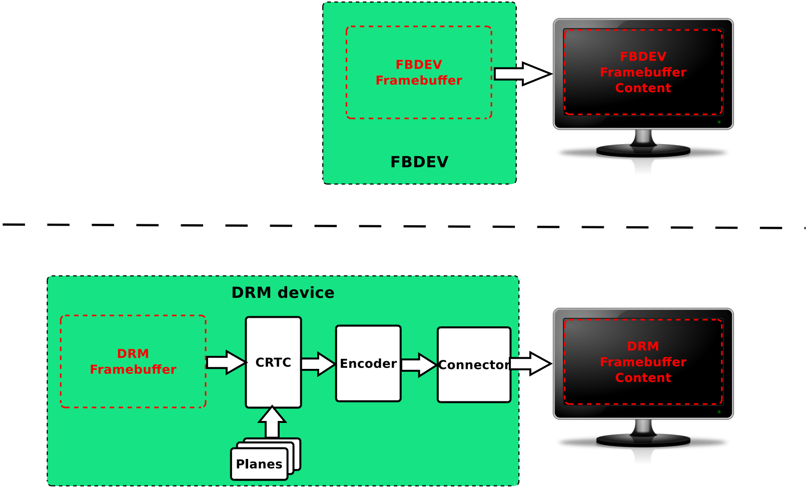
Context: Why choosing DRM/KMS?

- Actively maintained
- Provides fine grained control on the display pipeline
- Widely used by user-space graphic stacks
- Provides a full set of advanced features
- Why not FBDEV?
  - Less actively maintained
  - Does not provides all the features we needed (overlays, hw cursor, ...)
  - Developers are now encouraged to move to DRM/KMS
- Why not V4L2?
  - Well suited for video capture and specific video output devices but not for "complex" display controllers

DRM/KMS: Definition

- DRM stands for Direct Rendering Manager and was introduced to deal with graphic cards embedding GPUs
- KMS stands for Kernel Mode Setting and is a sub-part of the DRM API
- Though rendering and mode setting are now split in two different APIs (accessible through `/dev/dri/renderX` and `/dev/dri/controlDX`)
- KMS provide a way to configure the display pipeline of a graphic card (or an embedded system)
- KMS is what we're interested in when looking for an FBDEV alternative

DRM/KMS: Architecture



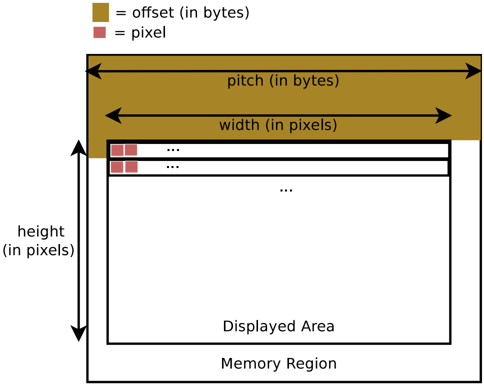
DRM/KMS Components: Framebuffer

- This is a standard object storing information about the content to be displayed
- Information stored:
  - References to memory regions used to store display content
  - Format of the frames stored in memory
  - Active area within the memory region (content that will displayed)

DRM/KMS Components: Framebuffer

- DRM Framebuffer is a virtual object (relies on a specific implementation)
- Framebuffer implementation depends on:
  - The memory manager in use (GEM or TTM)
  - The display controller capabilities:
    - Supported DMA transfer types (Contiguous Memory or Scatter Gather)
    - IOMMU
- Default implementation available for GEM objects using CMA (Contiguous Memory Allocator):
  `drivers/gpu/drm/drm_fb_cma_helper.c`
- Other implementations usually depend on the Display Controller
  - Scatter Gather example: `drivers/gpu/drm/tegra/`
  - IOMMU example: `drivers/gpu/drm/exynos/`

DRM/KMS Components: Framebuffer




```c
struct drm_framebuffer {
[...]
  unsigned int pitches[4];
  unsigned int offsets[4];
  unsigned int width;
  unsigned int height;
[...]
};
```

DRM/KMS Components: Framebuffer

```
struct drm_framebuffer {
[...]
  uint32_t pixel_format; /* fourcc format */
[...]
};
```

- `pixel_format` describes the memory buffer organization
- Uses FOURCC format codes
- Supported formats are defined here: `include/drm/drm_fourcc.h`
- These FOURCC formats are not standardized and are thus only valid within the DRM/KMS subsystem

DRM/KMS Components: Framebuffer

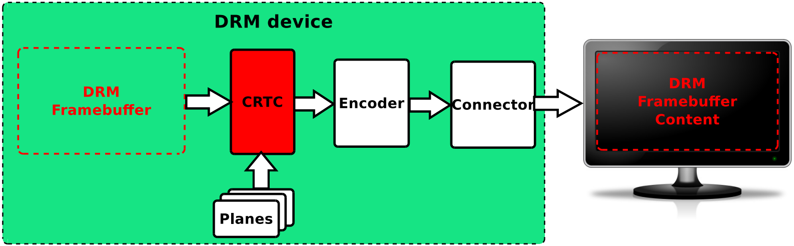
- Three types of formats used by the DRM/KMS subsystem:
  - RGB: Each pixel is encoded with an RGB tuple (a specific value for each component)
  - YUV: Same thing but with Y, U and V components
  - C8: Uses a conversion table to map a value to an RGB tuple
- YUV support different modes:
  - Packed: One memory region storing all components (Y, U and V)
  - Semiplanar: One memory region for Y component and one for UV components
  - Planar: One memory region for each component
- Each memory region storing a frame component (Y, U or V) is called a *plane*

DRM/KMS Components: Framebuffer

- Packed formats: only the first offsets and pitches entries are used
- Semiplanar formats: the first two entries are used
- Planar: the first 3 entries are used
- Don't know what the fourth entry used is for (alpha plane?)

```
struct drm_framebuffer {
[...]
  unsigned int pitches[4];
  unsigned int offsets[4];
[...]
};
```

DRM/KMS Components: CRTC



DRM/KMS Components: CRTC

- CRTC stands for CRT Controller, though it's not only related to CRT displays
- Configure the appropriate display settings:
  - Display timings
  - Display resolution
- Scan out frame buffer content to one or more displays
- Update the frame buffer
- Implemented through `struct drm_crtc_funcs` and `struct drm_crtc_helper_funcs`

```
struct drm_crtc_funcs {
[...]
  int (*set_config)(struct drm_mode_set *set);
  int (*page_flip)(struct drm_crtc *crtc,
           struct drm_framebuffer *fb,
           struct drm_pending_vblank_event *event, uint32_t flags);
[...]
};
```

DRM/KMS Components: CRTC (mode setting)

- `set_config()` is responsible for configuring several things:
  - Update the frame buffer being scanned out
  - Configure the display mode: timings, resolution, ...
  - Attach connectors/encoders to the CRTC
- Use `drm_crtc_helper_set_config()` function and implement `struct drm_crtc_helper_funcs` unless you really know what you're doing

```
struct drm_crtc_helper_funcs {
[...]
  int (*mode_set)(struct drm_crtc *crtc,
                  struct drm_display_mode *mode,
                  struct drm_display_mode *adjusted_mode,
                  int x, int y,
                  struct drm_framebuffer *old_fb);
[...]
};
```

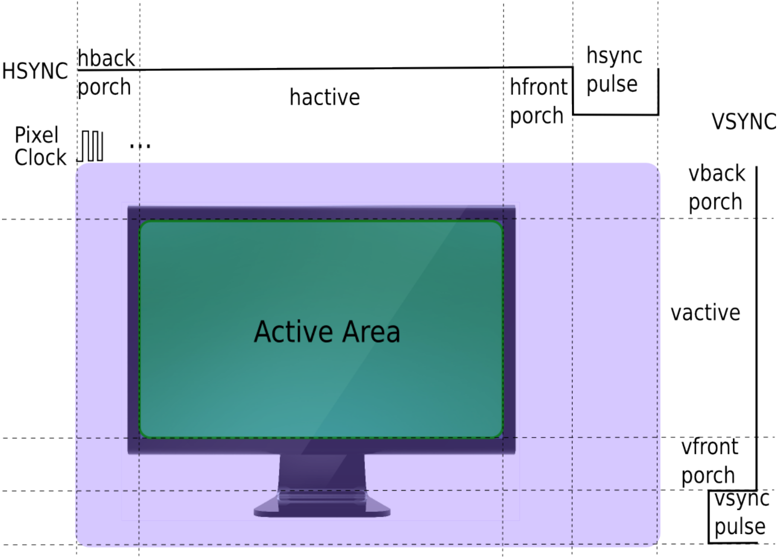
DRM/KMS Components: CRTC (display timings)

- How display content is updated hasn't changed much since the creation of CRT monitors (though technology has evolved)
- Requires at least 3 signals:
  - Pixel Clock: drive the pixel stream (1 pixel updated per clock cycle)
  - VSYNC: Vertical Synchronisation signal, asserted at the beginning of each frame
  - HSYNC: Horizontal Synchronisation signal, asserted at the beginning of each pixel line

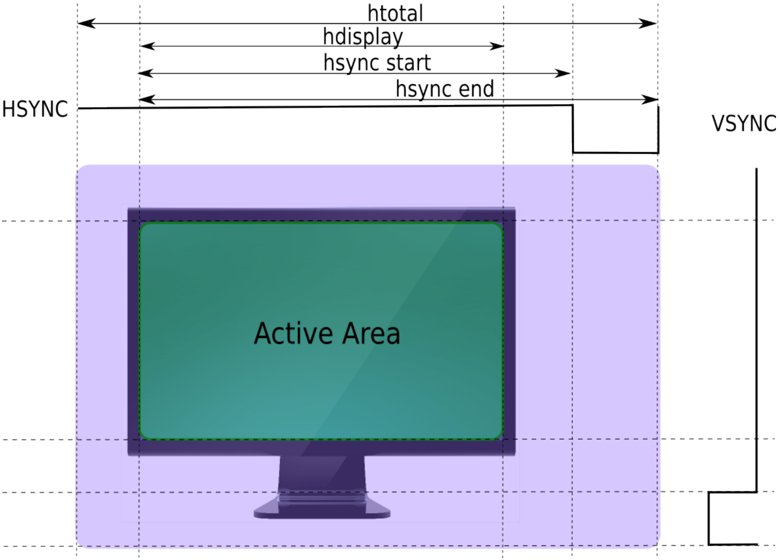
DRM/KMS Components: CRTC (display timings)

- HSYNC pulse is used to inform the display it should go to the next pixel line
- VSYNC pulse is used to inform the display it should start to display a new frame and thus go back to the first line
- What's done during the VSYNC and HSYNC pulses depends on the display technology
- Front and back porch timings are reserved time around the sync pulses. Action taken during these periods also depends on the display technology

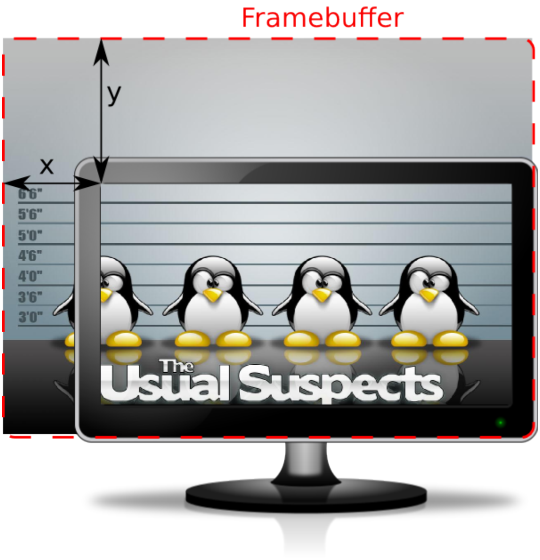
DRM/KMS Components: CRTC (display timings)



DRM/KMS Components: CRTC (display timings)



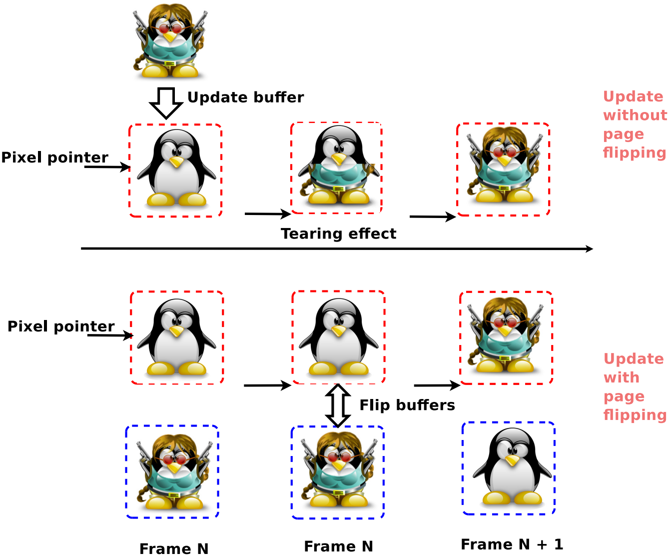
DRM/KMS Components: CRTC (mode setting)



DRM/KMS Components: CRTC (mode setting)

```
static int atmel_hlcdc_crtc_mode_set(struct drm_crtc *c,
                                     struct drm_display_mode *mode,
                                     struct drm_display_mode *adj,
                                     int x, int y,
                                     struct drm_framebuffer *old_fb)
{
  /* Initialize local variables */
  struct atmel_hlcdc_crtc *crtc = drm_crtc_to_atmel_hlcdc_crtc(c);
  [...]

  /* Do some checks on the requested mode */
  if (atmel_hlcdc_dc_mode_valid(crtc->dc, adj) != MODE_OK)
    return -EINVAL;

  /* Convert DRM display timings into controller specific ones */
  vm.vfront_porch = adj->crtc_vsync_start - adj->crtc_vdisplay;
  [...]

  /* Configure controller timings */
  regmap_write(regmap, ATMEL_HLCDC_CFG(1), (vm.hsync_len - 1) | ((vm.vsync_len - 1) << 16));
  [...]

  /* Update primary plane attached to the CRTC */
  fb = plane->fb;
  plane->fb = old_fb;

  return plane->funcs->update_plane(plane, c, fb, 0, 0, adj->hdisplay, adj->vdisplay,
                                    x << 16, y << 16, adj->hdisplay << 16,
                                    adj->vdisplay << 16);
}
```

DRM/KMS Components: CRTC (page flipping)



DRM/KMS Components: CRTC (page flipping)

- `page_flip()` is responsible for queueing a frame update

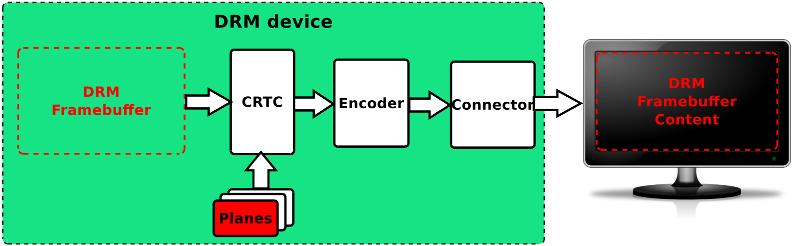
```
struct drm_crtc_funcs {
[...]
  int (*page_flip)(struct drm_crtc *crtc,
                   struct drm_framebuffer *fb,
                   struct drm_pending_vblank_event *event,
                   uint32_t flags);
[...]
};
```

- The frame is really updated at the next VBLANK (interval between 2 frames)
- Only one page flip at a time
- Should return `-EBUSY` if a page flip is already queued
- `event` is used to inform the user when page flip is done (the 2 frames are actually flipped)

DRM/KMS Components: CRTC (page flipping)

```
static int atmel_hlcdc_crtc_page_flip(struct drm_crtc *c, struct drm_framebuffer *fb,
                                      struct drm_pending_vblank_event *event,
                                      uint32_t page_flip_flags)
{
  /* Initialize local variables */
  struct atmel_hlcdc_crtc *crtc = drm_crtc_to_atmel_hlcdc_crtc(c);
  [...]

  /* Check if a there's a pending page flip request */
  spin_lock_irqsave(&dev->event_lock, flags);
  if (crtc->event)
    ret = -EBUSY;
  spin_unlock_irqrestore(&dev->event_lock, flags);
  if (ret)
    return ret;
  [...]

  /* Store the event to inform the caller when the page flip is finished */
  if (event) {
    drm_vblank_get(c->dev, crtc->id);
    spin_lock_irqsave(&dev->event_lock, flags);
    crtc->event = event;
    spin_unlock_irqrestore(&dev->event_lock, flags);
  }

  /* Queue a primary plane update request */
  ret = atmel_hlcdc_plane_apply_update_req(plane, &req);
  [...]

  return ret;
}
```

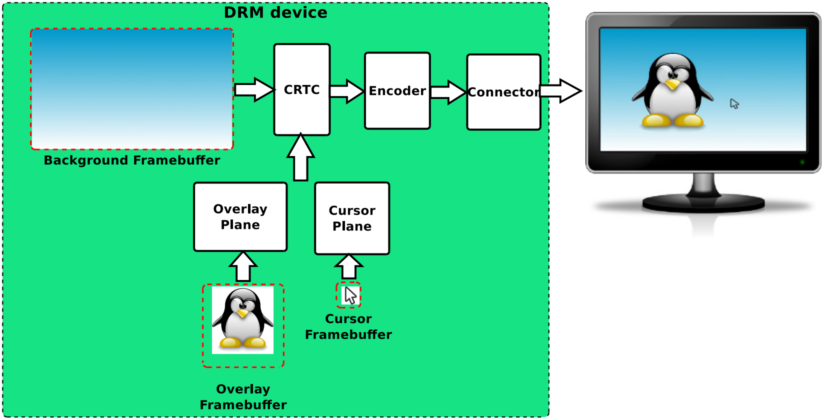
DRM/KMS Components: Planes



DRM/KMS Components: Planes

- A plane is an image layer (**Be careful**: not related to the planes referenced by a framebuffer)
- The final image displayed by the CRTC is the composition of one or several planes
- Different plane types:
  - `DRM_PLANE_TYPE_PRIMARY` (mandatory, 1 per CRTC)
    - Used by the CRTC to store its frame buffer
    - Typically used to display a background image or graphics content
  - `DRM_PLANE_TYPE_CURSOR` (optional, 1 per CRTC)
    - Used to display a cursor (like a mouse cursor)
  - `DRM_PLANE_TYPE_OVERLAY` (optional, 0 to N per CRTC)
    - Used to benefit from hardware composition
    - Typically used to display windows with dynamic content (like a video)
    - In case of multiple CRTCs in the display controller, the overlay planes can often be dynamically attached to a specific CRTC when required

DRM/KMS Components: Planes

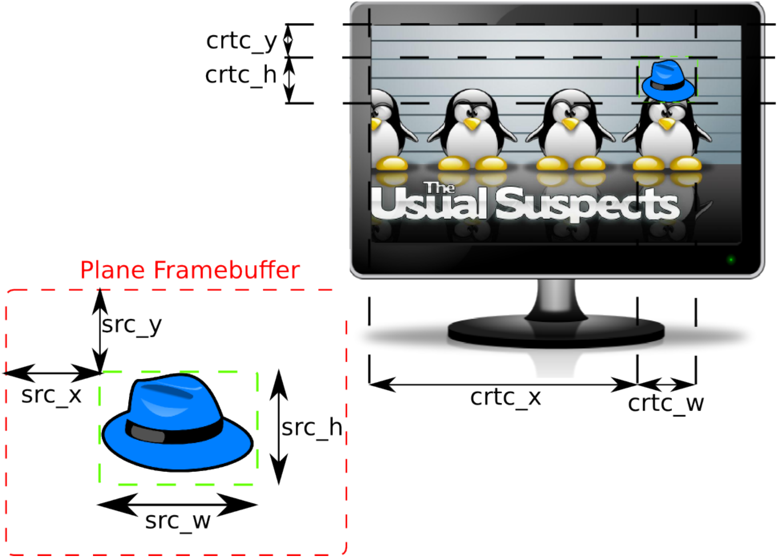


DRM/KMS Components: Planes

- Plane support implemented through `struct drm_plane_funcs`

```
struct drm_plane_funcs {
[...]
  int (*update_plane)(struct drm_plane *plane,
                      struct drm_crtc *crtc,
                      struct drm_framebuffer *fb,
                      int crtc_x, int crtc_y,
                      unsigned int crtc_w, unsigned int crtc_h,
                      uint32_t src_x, uint32_t src_y,
                      uint32_t src_w, uint32_t src_h);
[...]
};
```

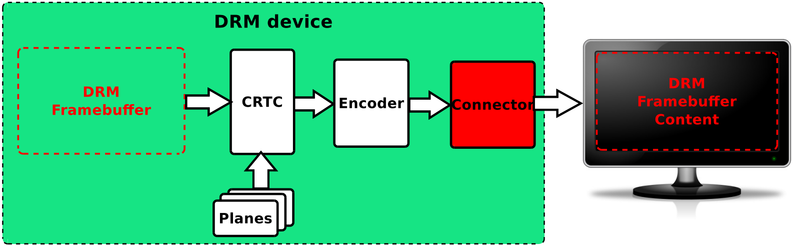
DRM/KMS Components: Planes (update)



DRM/KMS Components: Planes (update)

```
static int atmel_hlcdc_plane_update(struct drm_plane *p,
                                    struct drm_crtc *crtc,
                                    struct drm_framebuffer *fb,
                                    int crtc_x, int crtc_y,
                                    unsigned int crtc_w, unsigned int crtc_h,
                                    uint32_t src_x, uint32_t src_y,
                                    uint32_t src_w, uint32_t src_h)
{
  struct atmel_hlcdc_plane *plane = drm_plane_to_atmel_hlcdc_plane(p);
  struct atmel_hlcdc_plane_update_req req;
  int ret = 0;

  /* Fill update request with informations passed in arguments */
  memset(&req, 0, sizeof(req));
  req.crtc_x = crtc_x;
  req.crtc_y = crtc_y;
  [...]

  /* Prepare a plane update request: reserve resources, check request
     coherency, ... */
  ret = atmel_hlcdc_plane_prepare_update_req(&plane->base, &req);
  if (ret)
    return ret;
  [...]

  /* Queue the plane update request: update DMA transfers at the next
     VBLANK event */
  return atmel_hlcdc_plane_apply_update_req(&plane->base, &req);
}
```

DRM/KMS Components: Connector



DRM/KMS Components: Connector

- Represent a display connector (HDMI, DP, VGA, DVI, ...)
- Transmit the signals to the display
- Detect display connection/removal
- Expose display supported modes

DRM/KMS Components: Connector

- Implemented through `struct drm_connector_funcs` and `struct drm_connector_helper_funcs`

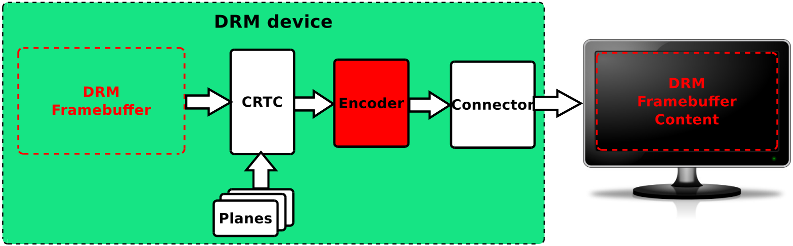
```
struct drm_connector_helper_funcs {
  int (*get_modes)(struct drm_connector *connector);
  enum drm_mode_status
      (*mode_valid)(struct drm_connector *connector,
                    struct drm_display_mode *mode);
  struct drm_encoder *
      (*best_encoder)(struct drm_connector *connector);
};
```

```
struct drm_connector_funcs {
[...]
  enum drm_connector_status
      (*detect)(struct drm_connector *connector, bool force);
[...]
};
```

DRM/KMS Components: Connector (get modes)

```
static int rcar_du_lvds_connector_get_modes(struct drm_connector *connector)
{
  struct rcar_du_lvds_connector *lvdscon = to_rcar_lvds_connector(connector);
  struct drm_display_mode *mode;

  /* Create a drm_display_mode */
  mode = drm_mode_create(connector->dev);
  if (mode == NULL)
    return 0;

  /* Fill the mode with the appropriate timings and flags */
  mode->type = DRM_MODE_TYPE_PREFERRED | DRM_MODE_TYPE_DRIVER;
  mode->clock = lvdscon->panel->mode.clock;
  mode->hdisplay = lvdscon->panel->mode.hdisplay;
  [...]

  /* Give this name a name based on the resolution: e.g. 800x600 */
  drm_mode_set_name(mode);

  /* Add this mode to the connector list */
  drm_mode_probed_add(connector, mode);

  /* Return the number of added modes */
  return 1;
}
```

DRM/KMS Components: Encoder



DRM/KMS Components: Encoder

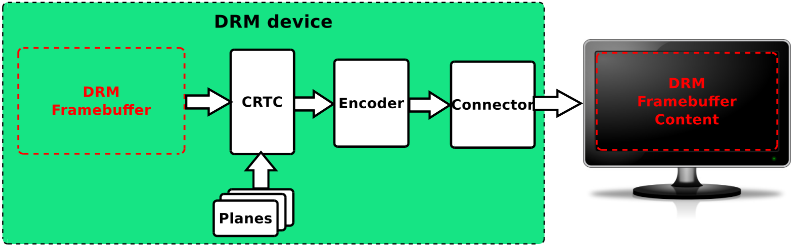
- Directly related to the Connector concept
- Responsible for converting a frame into the appropriate format to be transmitted through the connector
- Example: HDMI connector is transmiting TMDS encoded data, and thus needs a TMDS encoder.

DRM/KMS Components: Encoder

- Implemented through `struct drm_encoder_funcs` and `struct drm_encoder_helper_funcs`

```
struct drm_encoder_helper_funcs {
[...]
  bool (*mode_fixup)(struct drm_encoder *encoder,
                     const struct drm_display_mode *mode,
                     struct drm_display_mode *adjusted_mode);
[...]
  void (*mode_set)(struct drm_encoder *encoder,
                   struct drm_display_mode *mode,
                   struct drm_display_mode *adjusted_mode);
[...]
};
```

DRM/KMS Components: DRM device



DRM/KMS Components: DRM device

- Responsible for aggregating the other components
- Device exposed to userspace (handles all user-space requests)
- Implemented through `struct drm_driver`

```
struct drm_driver {
  int (*load) (struct drm_device *, unsigned long flags);
[...]
  int (*unload) (struct drm_device *);
[...]
  u32 driver_features;
[...]
};
```

DRM/KMS Components: DRM device

- Call `drm_dev_alloc()` then `drm_dev_register()` to register a DRM device
- `load()` and `unload()` are responsible for instantiating and destroying the DRM components attached to a DRM device
- `driver_features` should contain `DRIVER_RENDER`, `DRIVER_MODESET` or both depending on the DRM device features

DRM/KMS Components: DRM device

```
static struct drm_driver atmel_hlcdc_dc_driver = {
  .driver_features = DRIVER_HAVE_IRQ | DRIVER_GEM | DRIVER_MODESET,
  .load = atmel_hlcdc_dc_load,
  .unload = atmel_hlcdc_dc_unload,
[...]
  .name = "atmel-hlcdc",
  .desc = "Atmel HLCD Controller DRM",
  .date = "20141504",
  .major = 1,
  .minor = 0,
};
```

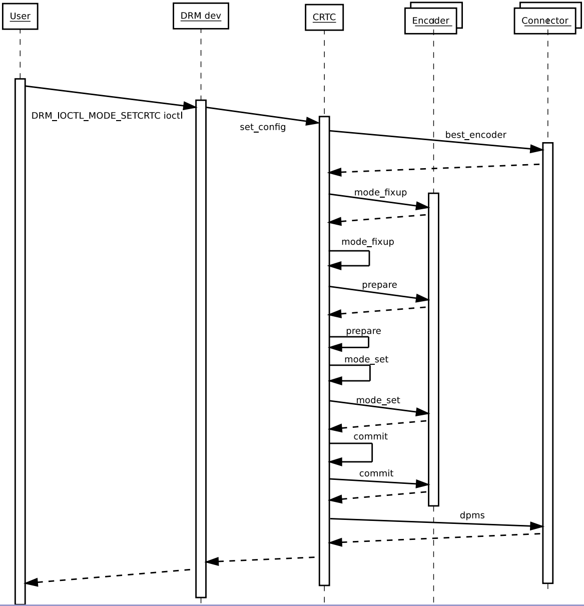
DRM/KMS Components: DRM device

```
static int atmel_hlcdc_dc_drm_probe(struct platform_device *pdev)
{
  struct drm_device *ddev;
  int ret;

  ddev = drm_dev_alloc(&atmel_hlcdc_dc_driver, &pdev->dev);
  if (!ddev)
    return -ENOMEM;

  ret = drm_dev_set_unique(ddev, dev_name(ddev->dev));
  if (ret) {
    drm_dev_unref(ddev);
    return ret;
  }

  ret = drm_dev_register(ddev, 0);
  if (ret) {
    drm_dev_unref(ddev);
    return ret;
  }

  return 0;
}
```

DRM/KMS Components: Other concepts

- DPMS: Display Power Management Signaling
- Properties: transversal concept used to expose display pipeline behaviors
  - Can be attached to all the components we've seen so far
  - Examples:
    - Rotation is a plane property
    - EDID (Unique display ID exposed by a monitor) is a connector property
    - ...
- Bridge: represents an external encoder accessible through a bus (i2c)
- Encoder slave: pretty much the same thing (still don't get the difference)
- FBDEV emulation
- Multiple CRTCs, Encoders and Connectors
- Other concepts I'm not aware of yet :-)

DRM/KMS Sequence Diagram: Mode Setting



KMS Driver: Development Tips

- Read the documentation: `gpu/drm-kms`
- Take a look at other drivers
  - Choose a similar driver (in terms of capabilities)
  - Check that the driver you are basing your work on is recent and well maintained
- Check for new features: the DRM subsystem is constantly evolving
- Use helper functions and structures as much as possible
- Start small/simple and add new features iteratively (e.g. only one primary plane and one encoder/connector pair)
- Use simple user-space tools to test it like `modetest`

KMS Userland Graphic Stacks

- Tried Weston (standard Wayland implementation) and Qt with a KMS backend
- First thing to note: they're not ready for KMS drivers without OpenGL support (`DRIVER_RENDER` capabilities)!
  - Wayland works (thanks to pixmam support) but does not support planes and hardware cursors when OpenGL support is disabled
  - Qt only works with the fbdev backend
  - WIP on the mesa stack to provide soft OpenGL when using a KMS driver without OpenGL support
  - But the window composition will most likely be done through the soft OpenGL, which implies poor performance
- Not sure you can choose a specific plane when using a window manager (e.g. stream video content on a plane which support YUV format)

Questions?

Boris Brezillon

boris.brezillon@bootlin.com

http://bootlin.com/pub/conferences/2014/elce/brezillon-drm-kms/