Adventure in DRMland
Or how to write a FreeBSD ARM64 DRM Driver

Emmanuel Vadot, manu@freebsd.org

Abstract

DRM (Direct Rendering Manager) is today standard for applications like a display server, to talk the the graphical hardware present on a computer or System On a Chip (SoC). It consist of a kernel side API and a userland part (via IOCTLs) that application can use to talk the GPU or configure the display modes (resolution, refresh rates etc ...).

While support for X86 devices (intel or amd) are now correct on FreeBSD, arm and arm64 hardware support is still lacking. The only DRM driver is for Tegra based SoC, other hardware either have basic framebuffer support (like the RaspberryPi family) or will require the bootloader to have framebuffer support and will use it via EFI framebuffer.

While framebuffer might be enough for some use, having a DRM driver brings a lots of possibility, 2D acceleration, changing resolution, hotpluging another monitor etc … And it is also mandatory if we want to support the 3D chip or the Video decoder usualy present on arm/arm64 SoCs.

In this paper the author will describe the DRM subsystem and the anatomy of a modern DRM driver, based on his work on the Allwinner Display Engine 2 present in many SoC of this semiconductor company.

1. Overview

1.1. DRM/KMS

DRM is an API first introduced to support 3D GPU in the Unix world. It only focused on GPU (command execution, textures etc …) and not on the mode setting part. A userland program (most likely X) used to do the mode setting by talking to the hardware directly to setup VGA/HDMI and all the rendering pipeline. This required X to run as root and also cause problems if multiple program wanted to configure the hardware.
To solve this a second API was later introduced called KMS (Kernel Mode Setting). The userspace didn't need to talk to the hardware anymore and will only need to call a few IOCTLs to configure the display. A few iteration of the API added more and more concept like framebuffers, planes, encoder etc …
This paper do not describe how to create a driver for a GPU but how to create a KMS driver. GPU on arm and arm64 are discrete ones and require a separate driver.

1.2. FreeBSD ARM{,64} Video Support

Support for video (in any possible form) on arm and arm64 on FreeBSD isn't available for a lot of platforms. The first driver was for the framebuffer on Raspberry Pi (r239922[i] by gonzo@ in August 2012), the bootloader/firmware on Rpi setup a display at startup and expose a framebuffer that the OS can use. To my knowledge it isn't possible to change the resolution or to setup the display after the OS is booted.
The second one was for the first generation of Allwinner Display Engine and supported only HDMI (r296064[ii] by jmcneill@ in February 2016). It worked by using modifed DTS files (file used to described the hardware on arm) and was later broken when we fully switched to the Linux DTS files.
In July of 2018 I added efifb support to ARM64 (r336520[iii]). As some bootloader supports video interface and EFI it allows us to have a working display with a simple driver. It has some limitation though, you cannot change the resolution or hotplug a monitor after that the OS is booted.

1.3. Chosen Hardware

To develop my first DRM driver I choose the Allwinner A64 SoC for multiple reasons :

1. This is a SoC that I know very well.
2. A lot of interesting hardware are available (or will be) like the PineBook and Pinetab ([iv])
3. Documentation, even if it isn't very explicit, exists. The only part not documented by Allwinner is the HDMI Transmitter but it is documented in NXP IMX.6 user manual.

2. KMS API and Object

2.1. Framebuffers and GEM Objects

2.1.1. Overview

Framebuffers are memory objects that holds the pixels needed to be rendered (or scanout) to the screen. They are composed of properties like width, height, pixels format etc … and between one and four GEM objects.
GEM Objects are the memory objects holding the pixel data, this is the hardest part of DRM (atleast for me) as it is directly tied to the VM subsystem.

2.1.2. Implementation

For embedded devices where you don't have graphic memory, one should use the gem_cma implementation (CMA: Contigous Memory Allocator).
The problem is that this part is gplv2 only. A BSD implementation based on NVIDIA Tegra DRM driver by mmel@ is currently being written by the author.

2.2. Planes

2.2.1. Overview

Planes object are backed by drm_framebuffer objects and are tied to a CRTC (see 2.3).
Multiple types of plane exists :

1. Primary plane is used for the main framebuffer so it will likely contain the pixel for your hole desktop environment. There can be only one per CRTC.
2. Cursor plane is used for mouse cursor. Some hardware can do composition directly (blending and mixing multiple buffers) this avoid doing that in software and waste CPU cycle. Cursor plane are usually small, 64x64 pixels is a common size.
3. Overlay planes are just generic plane that can be used for anything. One common application of them is for video, the video player will have his own plane where it can render data directly in it and the compositing hardware will handle the blending/mixing part.

In the Allwinner DE2 there is two units (called mixer) that handle the planes/overlays. One have three planes that support RGB format and one plane that support YUV format (for Video purpose) while the other one only have one RGB plane and one YUV plane.

2.2.2. Implementation

A drm driver can register plane using the *drm_universal_plane_init* function.

```
int drm_universal_plane_init(struct drm_device *dev,
                             struct drm_plane *plane,
                             uint32_t possible_crtcs,
                             const struct drm_plane_funcs *funcs,
                             const uint32_t *formats,
                             unsigned int format_count,
                             const uint64_t *format_modifiers,
                             enum drm_plane_type type,
                             const char *name, ...);
```

The helper funcs are fully provided by the KMS framework and so a driver can simply use the default one.

```
static const struct drm_plane_funcs plane_funcs = {
  .atomic_destroy_state = drm_atomic_helper_plane_destroy_state,
 .atomic_duplicate_state = drm_atomic_helper_plane_duplicate_state,
 .destroy = drm_plane_cleanup,
 .disable_plane = drm_atomic_helper_disable_plane,
 .reset = drm_atomic_helper_plane_reset,
 .update_plane = drm_atomic_helper_update_plane,
};
```

What a driver only need to implements are the helper funcs for atomic mode setting.

```
int plane_atomic_check(struct drm_plane *plane,
                       struct drm_plane_state *state)
void plane_atomic_disable(struct drm_plane *plane,
                          struct drm_plane_state *old_state)
void plane_atomic_update(struct drm_plane *plane,
                          struct drm_plane_state *old_state)

static struct drm_plane_helper_funcs plane_helper_funcs = {
            .atomic_check          = plane_atomic_check,
            .atomic_disable        = plane_atomic_disable,
            .atomic_update         = plane_atomic_update,
};

drm_plane_helper_add(struct drm_plane *plane, &plane_helper_funcs);
```

The atomic_check function will return 0 if the plane can be drawn by the hardware. Most of the time just calling *drm_atomic_helper_check_plane_state* is sufficient.
The atomic_disable function will disable the plane from being rendered.
The atomic_update function will update all the plane information (address, format, size etc …) in the hardware.

2.3. CRTCs

2.3.1. Overview

CRTC stand for Cathode Ray Tube Controller, this is an historical name and the KMS crtc object don't have anything to do with CRT monitor.
CRTC take the contents of the framebuffers and planes and output the final image on a physical bus. This can be an external bus (some RGB pins to drive a lcd panel for example) or an internal bus that goes into an HDMI or VGA encoder (see 2.4).

In Allwinner SoCs you have again two different units (called TCON), one can output pixel directly in RGB format, MIPI-DSI and LVDS while the other is directly tied to the HDMI transmitter. Both can take their inputs from either of the two mixers but the default configuration is that the mixer0 (The one with 4 planes) outputs to the TCON0 and mixer1 outputs to TCON1.

2.3.2. Implementation

The easiest way to register a crtc in the subsystem is with the function *drm_crtc_init_with_planes.*

```
int drm_crtc_init_with_planes(struct drm_device *dev,
                              struct drm_crtc *crtc,
                              struct drm_plane *primary,
                              struct drm_plane *cursor,
                              const struct drm_crtc_funcs *funcs,
                              const char *name, ...);
static const struct drm_crtc_funcs crtc_funcs = {
 .atomic_destroy_state  = drm_atomic_helper_crtc_destroy_state,
 .atomic_duplicate_state = drm_atomic_helper_crtc_duplicate_state,
 .destroy = drm_crtc_cleanup,
 .page_flip = drm_atomic_helper_page_flip,
 .reset = drm_atomic_helper_crtc_reset,
 .set_config = drm_atomic_helper_set_config,
 .enable_vblank = crtc_enable_vblank,
 .disable_vblank = crtc_disable_vblank,
};
```

In the crtc_funcs only two function for enabling/disabling vblank interrupt need to be implemented, for the others the helpers functions are enough.
Some helper functions for atomic mode setting are also needed :

```
static const struct drm_crtc_helper_funcs crtc_helper_funcs = {
            .atomic_check          = crtc_atomic_check,
            .atomic_begin          = crtc_atomic_begin,
            .atomic_flush          = crtc_atomic_flush,
            .atomic_enable         = crtc_atomic_enable,
            .atomic_disable        = crtc_atomic_disable,
            .mode_set_nofb         = crtc_mode_set_nofb,
};
drm_crtc_helper_add(crtc, &crtc_helper_funcs);
```

All of those functions need to be implemented but even if my current implementation seems to work I don't fully understand what they are really supposed to do. They deal with vblank and events and I am really not familiar with them enough.

2.4. Encoders

2.4.1. Overview

drm_encoder simply convert one pixel data bus format to another one. For example they can convert one internal format (such as between TCON1 and the HDMI transmitter) to TMDS, the signal format used in HDMI transmission.

2.4.2. Implementation

Only one helper function needs to be implemented for encoder : the mode_set one.
This is used to set the clock rate of the encoder at the same rate at the pixel clock for example.

Then using drm_encoder_helper_add and drm_encoder_init on can register the encoder in the DRM subsystem.

```
drm_encoder_helper_add(&sc->encoder,&encoder_helper_funcs);
sc->encoder.possible_crtcs = drm_crtc_mask(crtc);
drm_encoder_init(drm, &sc->encoder, &encoder_funcs,
DRM_MODE_ENCODER_TMDS, NULL);
```

2.5. Bridges/Connectors

2.5.1. Overview

drm_connector simply represent a physical connector on the card or single board computer.
drm_bridge sits between an encoder and a connector.
They are used to enable/disable the display and configuring the display mode (resolution and timing).

2.5.2. Implementation

For drm_connector only one function need to be implemented : connector_detect. This isn't possible for every connector type but for HDMI it is.

```
static enum drm_connector_status
connector_detect(struct drm_connector *connector, bool force);

static const struct drm_connector_funcs dw_hdmi_connector_funcs = {
 .fill_modes = drm_helper_probe_single_connector_modes,
 .detect = connector_detect,
 .destroy = drm_connector_cleanup,
 .reset = drm_atomic_helper_connector_reset,
 .atomic_duplicate_state = drm_atomic_helper_connector_duplicate_state,
 .atomic_destroy_state = drm_atomic_helper_connector_destroy_state,
};
```

One helper function is also needed, the get_modes one, it is called for quering EDID from the connected monitor.

```
static const struct drm_connector_helper_funcs
connector_helper_funcs = {
		.get_modes = connector_get_modes,
};
```

For the bridges the following functions need to be implemented :

```
static int
bridge_attach(struct drm_bridge *bridge);
static enum drm_mode_status
bridge_mode_valid(struct drm_bridge *bridge, const struct
drm_display_mode *mode);
static void
bridge_mode_set(struct drm_bridge *bridge,
  struct drm_display_mode *orig_mode,
  struct drm_display_mode *mode);
static void
bridge_disable(struct drm_bridge *bridge);
static void
bridge_enable(struct drm_bridge *bridge);

static const struct drm_bridge_funcs bridge_funcs = {
		.attach = bridge_attach,
		.enable = bridge_enable,
		.disable = bridge_disable,
		.mode_set = bridge_mode_set,
		.mode_valid = bridge_mode_valid,
};
```

bridge_attach needs to init and attach the connector.
bridge_mode_valid need to filter the mode
bridge_mode_set just need to copy the desired mode that will be used in the enable function.

3. Current status and Future work

3.1. Current status

As of 20180225, my current implementation support the mixer1 and tcon1 IP block so only HDMI output is currently possible. There is still problem in the HDMI driver to do a full bring-up, it doesn't work yet if u-boot is configured without video support, this is probably just a few registers that aren't setup correctly.

3.2. Future work

Fixing all the bugs and testing different monitors (with differents supported resolutions) is my main priority.
Next I will finish the BSD implementation of the CMA function.
Adding support for other Allwinner SoC (such as the arm32 H3 and arm64 H5) is also planned in the near future.
LIMA (The reversed-engineered MALI driver) will be very interesting to have in FreeBSD.

i https://svnweb.freebsd.org/base?view=revision&revision=239922
ii https://svnweb.freebsd.org/base?view=revision&revision=296064
iii https://svnweb.freebsd.org/base?view=revision&revision=336520
iv https://www.pine64.org/