DOI: 10.17725/rensit.2021.13.087

Software architecture for display controller and operating system interaction

Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov

Center of Visualization and Satellite Information Technologies, Federal Scientific Center Scientific Research Institute of System Analysis of the RAS, https://niisi.ru/

36/1, Nachimovsky av., Moscow 117218, Russian Federation

E-mail: rilian@niisi.ras.ru, mamrosenko_k@niisi.ras.ru, algts@niisi.ras.ru

Received January 15, 2021, peer-reviewed Jamuary 22, 2021, accepted January 29, 2021

Abstract: **Article describes solutions for developing programs that provide interaction between Linux operating system and multiple display controller hardware blocks (outputs), that use one clock generation IP-block with phase-locked loop (PLL). There is no API for such devices in Linux, thus new software model was developed. This model is based on official Linux GPU developer driver model, but was modified to cover case described earlier. Article describes three models for display controller driver development – monolithic, component and semi-monolithic. These models cannot cover case described earlier, because they assume that one clock generator should be attached to one output. A new new model was developed, that is based on component model, but has additional mechanics to prevent race condition that can happen while using one clock generator with multiple outputs. Article also presents modified model for bootloaders graphics drivers. This model has been simplified over developed Linux model, but also has component nature (with less components) and race prevention mechanics (but with weaker conditions). Hardware interaction driver components that are developed using provided software models are interchangeable between Linux and bootloader.**

Keywords: **drivers, embedded, KMS, kernel module, development**

UDC 004.454

Acknowledgments: Publication is made as part of national assignment for SRISA RAS (fundamental scientific research 47 GP) on the topic No.0580-2021-0001 (AAAA-A19-119011790077-1).

For citation: Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov. Software architecture for display controller and operating system interaction. *RENSIT*, 2021, 13(1):87-94. DOI: 10.17725/rensit.2021.13.087.

Contents

1. INTRODUCTION

In some cases, development of embedded systems requires compromise decisions to be made to secure compliance of the product with the applicable requirements. Such a situation can

be seen, for example, when restrictions apply to heat release or crystal size, or when transistor budget needs to be reduced.

Sometimes, such compromise solution is to reduce the number of frequency synthesizers (FS). Multiple display controllers use the same FS, but different display interface protocols (e.g. TMDS and LVDS) are used while the outputs operate concurrently.

In addition, such embedded systems can use a common configurable register. In such register, both general parameters of the whole display subsystem and some parameters of individual controllers are often configured. This complicates programming of the output subsystem, since such register becomes a shared resource and requires access control.

Such display system configuration requires new approaches to be developed to create drivers that ensure that the system is functional. Accordingly, our tasks are as follows:

1. Consider possible models for creation of drivers for such embedded systems;
2. Compare the models under consideration in terms of extensibility and functionality of the resulting drivers;
3. Develop a driver model that will allow solving the problem of interaction with the operating system when multiple display controllers and one frequency synthesizer are used.

The research will be conducted on the basis of Linux OS, which is a common operating system for embedded systems.

There are several kernel subsystems used in development of a display controller driver for Linux. The applied model depends on the subsystem used (which can be specified in documentation or identified from an analysis of sample drivers). Let us look at the most popular ones:

1. DRM. As stated in [1], the display controller and mode setting interface is part of the Direct Rendering Manager (DRM) infrastructure. DRM developers [2] consider this subsystem in terms of 2 models — an external and an internal ones. According to their views, the external model includes 5 entities, such as framebuffer, plane, CRTC, encoder and connector, which are listed from proximity to the user space to proximity to the display. The internal subsystem model additionally includes a bridge between the encoder and the connector, while the connector can also include a panel. In contrast, software developers [3] and independent researchers [4] generally consider an external model or a simplified external model only.
2. FrameBuffer device (fbdev). If fbdev [4] is used, the model consists of 2 components: a memory buffer interacting with the user space and a monolithic virtual DC (display controller) device that controls several physical devices. The modes of the device were set directly from the user space thus causing problems [5]. Along with the identified problems, such simplified model has low extensibility and lower functionality than models based on the DRM subsystem [6]. Currently, the latest available drivers based on this model are being redeveloped into drivers based on DRM model variations [7]
3. Driver solutions for Android Display Framework (ADF) exist, but have the following disadvantage: standard hardware-independent protocol functions need to be developed from scratch in each driver increasing the possibility of errors and the complexity of the task. In addition, as the ADF subsystem is a development of fbdev, drivers need to be combined, as despite the atomicity, models based on this subsystem remain monolithic, which was the reason for replacement of the subsystem with DRM [8].

2. MATERIALS AND METHODS. GRAPHICS DRIVER DEVELOPMENT MODELS USING THE LINUX DRM SUBSYSTEM

Controller driver solutions in a number of other Unix-compatible systems (e.g. FreeBSD) also use the DRM subsystem as the most extensible and functional open subsystem for development of such drivers [9].

To develop a driver using the DRM, an approach should be used so that the driver model is not in conflict with the external and internal subsystem models, thus allowing the use of the DRM API for programming display controllers, especially when an image is displayed by multiple display units simultaneously. Three models were analyzed during the development of the driver.

A completely monolithic type (**Fig. 1**) is characterized by relative simplicity of the device, but at the same time it has a very high cohesion. This implies that various DTLC (display transmitter link controller – a controller that encodes and decodes signals for a display unit and works with a graphics transmission protocol) subsystems and various FS types require writing of completely different drivers, despite an identical IP-block of the display controller. Despite that such driver formally belongs to the

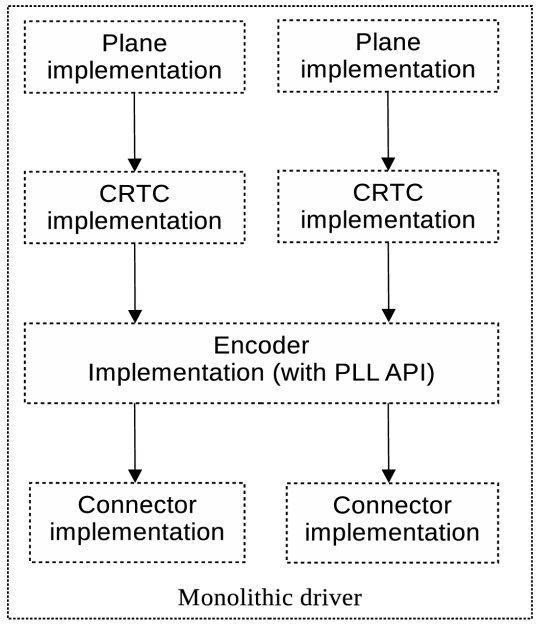

Fig. 1. *Monolithic driver model.*

DRM subsystem, the model has the advantages and disadvantages of the previous generation subsystems. This significantly limits the ability of the drivers based on such model to update and replace components – a new driver will have to be developed for each revision of the system. An advantage to be noted is the simplicity of the driver for each device.

The second type is a cohesive component model considering possible replacement of hardware (**Fig. 2**). Drivers based on this model have a slightly lower component

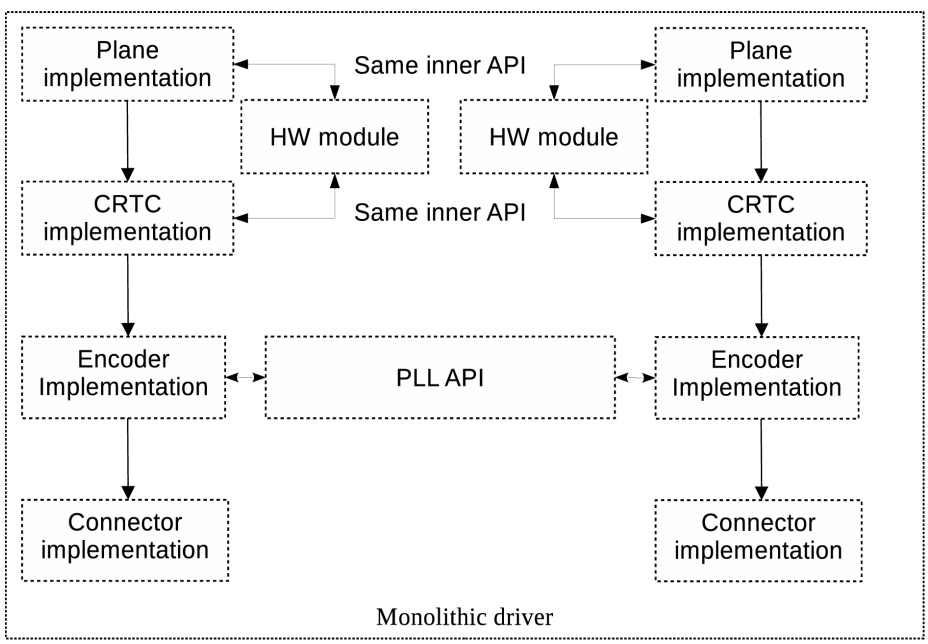

Fig. 2. *Semi-monolithic driver model.*

cohesion, which simplifies modification for a new controller type, but does not solve the problem of extensibility and adds complexity to the driver due to separation of components. Despite internal division into components and lower cohesion, drivers based on this model still have an important disadvantage of completely monolithic drivers, such as the need to modify the whole driver for each combination of components in all possible revisions, which complicates driver unification.

The third type is built on the basis of the external DRM model and following the example of other drivers for embedded systems that use the DRM subsystem [10] – a model that is developed on the principle of maximum independence and interchangeability of components (**Fig. 3**). Drivers based on this model are extensible and all their components are interchangeable. For example, it is possible to replace only a DTLC driver, or only a frequency synthesizer driver, or only a configuration device driver. It is also possible to use drivers that have been already developed (with minor modifications for integration), which is important when IP-blocks are purchased from other manufacturers. In addition, as the model is not monolithic, it is possible to build an API that will address common devices avoiding the race condition and, at the same time, will not integrate the driver of the device into a monolithic structure.

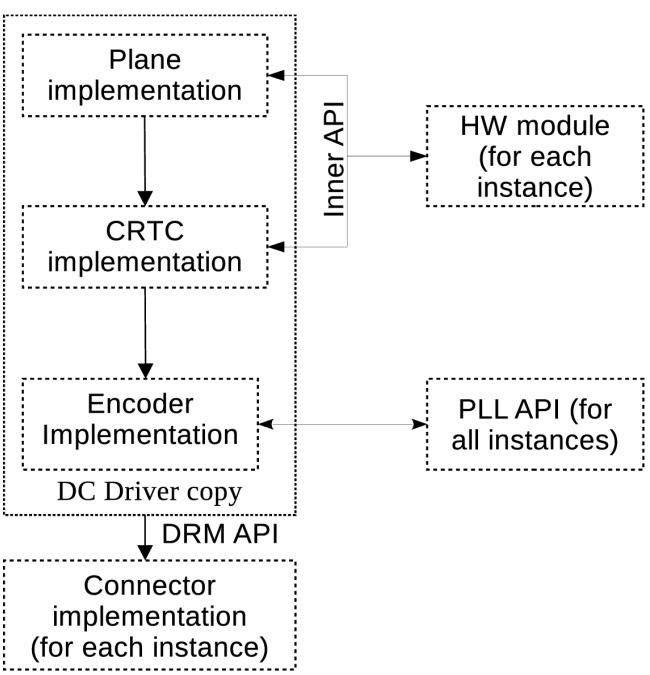

Fig. 3. *Component driver model.*

When only the external model is considered and all the components have maximum independence, a more complex algorithm than the one implied by abstraction of the external model is required to control some DTLC controllers. For example, interaction with the DTLC component has to be taken into account when the controller is switched on and off.

Since many modern display interface protocols (e.g. DisplayPort) require the use of the above algorithm, the presented model needs to be modified considering their features.

3. RESULTS. DEVELOPMENT OF A NEW DRIVER MODEL FOR A DEVICE WITH SEVERAL OUTPUT DEVICES AND ONE FREQUENCY SYNTHESIZER

Among the solutions described above, the component solution was tested (Fig. 3) as the most extensible one, which uses the internal mechanisms of the Linux kernel to a greater extent. During testing of the driver based on solution 3, this solution revealed the following disadvantages:

1. Race condition when the clock generator is addressed. Since all instances work with the clock frequency control API, when the FS is addressed in parallel the frequency will vary in a chaotic manner, which will lead to a flickering screen and incorrect operation of an output.
2. When only the external DRM model is used, it is impossible to create such DTLC driver that requires complex interaction with the FS or display controller driver. In the external DRM model, the replaceable component is the connector, which does not have

dynamic APIs for interacting with DTLC [2]. Therefore, an internal DRM model has to be used, which is incompatible with the model shown in Fig. 3.

3. Driver instances within the kernel based on the component model are completely independent, which makes it difficult to transfer data between them. Since the instances are identical, a race condition can occur when data are written in the configuration register.

To solve the above problems, modifications were made in the component model, which eventually became as follows (**Fig. 4**):

1. The first instance of the DC driver becomes the only instance that is able to transmit information to the FS driver. All instances are able to receive information.
2. The first instance of the driver determines configuration parameters that are common for all instances and sets the same in the common configuration register.
3. To link the display controller driver and the DTLC driver, the bridge component of the internal DRM model is used, which operates jointly with the connector component of the external model.

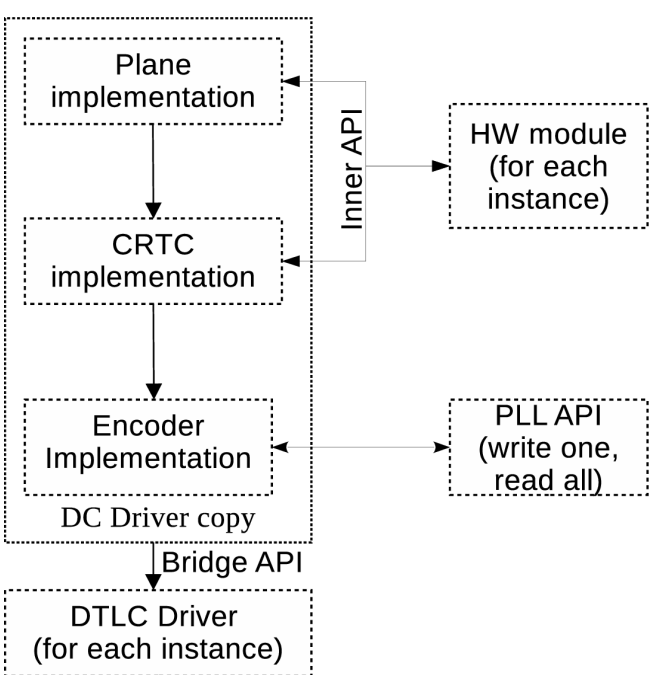

Fig. 4. *Newly developed component-based model.*

4. Parameters of permissible modes are calculated as:

$$a_1 \cdot x_1 - a_2 \cdot x_2 \leq b,$$

where a_i is the coefficients associated with DTLC (the exact value depends on the DTLC hardware), b is the error coefficient (which depends on the interface protocols; in particular, for DVI the coefficient is 0.025[11]), and x_i is the pixel frequency coefficients of the requested modes. In this case, ai and b coefficients are parameters that are set when a driver based on the model shown in Fig. 4 is developed, while xi is the variables calculated when the driver is running.

Most systems working under Linux use low-level bootloaders (U-Boot [12], BareBox [13]) for initial initialization and loading of devices, including display subsystem devices (**Fig. 5**). In such cases, apart from a driver for the OS, a bootloader driver that is capable of performing the basic display functions also has to be developed. With some modifications in the model due to simplified nature of the graphic API of the bootloader (if any) and incomplete suitability of the graphic API for the DRM subsystem, parts of the driver for Linux based on the developed model can be used. To be able to use low-level bootloaders (in particular, those listed above) in the drivers, the following modifications were made in the model:

1. There is no separate DTLC module in the bootloader, as no work with complex

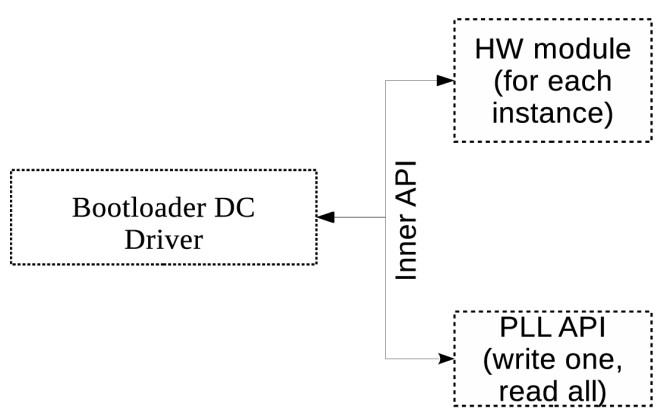

Fig. 5. *Bootloader driver model*

protocols is carried out. Therefore, everything is included in the display controller driver.

2. The DC driver model in the bootloader does not have such entities as plane, encoder and crtc. All display operations are carried out in the same structure, so the DC driver is monolithic. Despite being similar to the model based on fbdev, the bootloader subsystem has fewer features. This is due to the bootloader's focus on quick booting of the operating system and reduction of the bootloader size.
3. The bootloader does not require simultaneous use of several screens, so one instance of the display driver is sufficient in case of identical output devices.

Simultaneous use of the FS by multiple display devices is very rarely met in the graphic subsystem of the bootloader, so the only remaining option is simultaneous use of the same memory area by multiple devices. Consequently, drivers are significantly simplified.

An interesting point is also the relationship between the bootloader driver components and the OS driver. If the OS driver is based on the model described in the previous chapter, its module for interaction with the hardware will be applicable in the bootloader if the internal API is the same. This significantly simplifies writing of complex device drivers, where interaction with hardware requires algorithms for calculation of parameters (independent of the high level) or sequences of data records in the device registers.

Despite a simpler structure, bootloader drivers inherit the component principle. This allows an incremental update of driver components both in case of a change in the hardware and when the bootloader is modified making the update easier [14].

4. DISCUSSION

The developed model has the following advantages over the considered model options:

1. Extensibility – the DC driver based on this model can be extended with any DTLC driver (including those developed before creation of a driver based on the above model) that uses bridge API.
2. Accelerated transition to new controller models. As transition to a new controller model requires only development of a new module for interaction with hardware, no complete redevelopment of the driver is required in comparison with the model shown in Fig. 1, and no complete update of the driver (it is enough to update the module for interaction with hardware only) is required in comparison with model 2.
3. Unlike drivers based on the component model, as shown in Fig. 3, drivers based on the developed model do not go into the race condition if the application software constantly requests the DRM API to change the frequency.
4. Furthermore, unlike the model shown in Fig. 3, drivers based on the modified model are able to display multiple images on all display units at the same time (when the permissible modes are set). The driver based on model 3 will give an unstable image in case of constant recording in the FS (as different display units require different pixel frequencies, the image will flicker), and drivers based on models 1 and 2 will set the frequency of the display unit, which was connected last).

In comparison with the considered models, the developed model has the following disadvantages:

1. Increased complexity of driver development in comparison to model 1.
2. More failure possibilities due to a larger number of points of failure (more driver instances and possibilities of random recombination with DTLC drivers).
3. The use of Bridge API is complicated in architectures with no device tree; complex

data manipulations to find the bridge are needed.

5. CONCLUSION

In the course of the research, various possible variants of the driver architecture for a device with multiple display controllers and one frequency synthesizer were analyzed. From several known driver models, a new derived driver model for such a device was developed, which combines the advantages of all the proposed models, but has a disadvantage – creation of a driver based on this model is difficult if no device-tree is used. This disadvantage is not significant for embedded systems with Linux, as most of them use a device tree.

The created model was tested during the development of the driver for a device with DVI outputs, LVDS on a single FS, under Linux, with MIPS architecture. All model implementations were successfully tested using the Protium prototyping system [15].

The results of the research can be used to create new drivers for this class of devices for Unix-like systems if hardware with multiple outputs and one frequency synthesizer is used.

REFERENCES

1. Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov. Visualization of graphic information in general-purpose operating systems. Radioelektronika, Nanosistemy, Informacionnye Tehnologii. (RENSIT), 2019, 11(2):217-224. DOI: 10.17725/rensit.2019.11.217.
2. Linux GPU Driver Developer's Guide [Electronic resource]. 2019. URL: https://dri.freedesktop.org/docs/drm/gpu/index.html (accessed: 06.03.2019).
3. Rob Clark. GStreamer and dmabuf. *GStreamer Conference*. San Diego, USA: Linux Foundation, 2012.
4. Laurent Pinchart. DRM/KMS, FB and V4L2: How to Select a Graphics and Video API. *Embedded Linux Conference Europe*. Barcelona, Spain: Linux Foundation, 2012.
5. Luc Verhaegen. X and Modesetting: Atrophy illustrated. *FOSDEM*, 2006, Brussels, Belgium, 2006.
6. Laurent Pinchart. Why and How to use KMS as Your Userspace Display API of Choice. *LinuxCon*. Tokyo, Japan, 2013, p. 52.
7. Michael Larabel. SUSE Develops New Driver That Exposes DRM Atop FBDEV Frame-Buffer Drivers [Electronic resource]. 2019. URL: https://www.phoronix.com/scan.php?page=news_item&px=FBDEVDRM-DRM-Over-FBDEV (accessed: 22.06.2019).
8. Alistair Strachan. DRM/KMS for Android. *Linux Plumbers*. Vancouver, BC: Linux Foundation, 2018.
9. Jean-Sébastien Pédron. Status of the Graphics Stack on FreeBSD. *X.Org Developer's Conference*. Bordeaux, France, 2014.
10. Marek Szyprowski. Linux DRM: New picture processing API. *LinuxCon.,* Berlin, Germany: Linux Foundation, 2016.
11. Digital Display Working Group. Digital Visual Interface DVI Revision 1.0 [Electronic resource]. 1999. URL: https://web.archive.org/web/20120813201146/http://www.ddwg.org/lib/dvi_10.pdf (accessed: 30.12.2020).
12. Anatolij Gustschin. U-Boot video API [Electronic resource]. 2020. URL: git https://gitlab.denx.de/u-boot/custodians/u-boot-video.git (accessed: 29.07.2020).
13. Hauer S. BareBox [Electronic resource]. 2020. URL: https://github.com/saschahauer/barebox (accessed: 29.07.2020).
14. Y. Kang, J. Chen, B. Li. Generic Bootloader Architecture Based on Automatic Update Mechanism. *IEEE 3rd International Conference*

on Signal and Image Processing (ICSIP). Shenzhen, China, 2018, p. 586-590.

15. Andrey Y. Bogdanov. Experience in the Use of Protium FPGA-Based Prototyping Platform to Verify Microprocessors. *SRISA RAS Papers*, 2017, 7(2):46-49.