VISUALIZATION OF GRAPHIC INFORMATION IN THE GENERAL-PURPOSE OPERATING SYSTEMS

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

Scientific Research Institute of System Analysis of the RAS, Center of Visualization and Satellite Information Technologies, https://niisi.ru
Moscow 117218, Russian Federation
rilian@niisi.ras.ru, mamrosenko_k@niisi.ras.ru, algts@niisi.ras.ru

Abstract. **Article describes solutions for developing programs for interaction between Linux operating system and display controller. Operating system architecture encourages creating a driver — component, which task is to perform interaction of hardware controller and OS kernel with the use of many protocols. The development of drivers for the open source OS is difficult due to continuous changes in the structure of the kernel, which breaks backward compatibility frequently. Several approaches to display controller driver development are provided in the article. Basic concepts of these drivers include Kernel Mode Setting (or KMS), meant to provide driver in the kernel and User Mode Setting, meant to provide driver in graphical server. Specific way is used for develop Linux kernel drivers - it is half-procedural, and half-object oriented. Sometimes it is called as "OOP in C", because it is based on GNU C Language extensions usage to simulate object-oriented techniques. GNU C usage almost prevents using non-GNU compilers to build a driver. Newest concept of writing display controller driver for Linux kernel is based upon atomic mode setting concept, in modern Linux it is achieved via state concept - intermediate states of an object are stored and modified without side effects to other objects. In order to make this concept work, driver should conform to the principle "disable function undoes all effects of enable and nothing more". Several approaches for display driver development are provided in this article, and most modern method – atomic KMS mode setting, - is described in detail.**

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

UDC 004.454

Bibliography - 11 references *Received 23.01.2019, accepted 05.02.2019*

RENSIT, 2019, 11(2):217-224 **DOI: 10.17725/rensit.2019.11.217**

Contents

1. Introduction (217)
2. Components of the Graphic System (218)
3. Linux Graphic Systems and Drivers (2182)
4. Kernel Mode Setting and User Mode Setting. Historical Background (219)
5. Object Model of the Linux Kernel (220)
6. KMS Atomic Drovers. Basic Entities and Interfaces Necessary for Implementation (221)
7. Debugging Driver (222)
8. Conclusion (223)

References (223)

1. INTRODUCTION

Currently, due to the increase in the share of various wearable and embedded systems, it is a relevant objective to develop drivers for them. Many embedded systems use the Linux OS, which has the support of a large number of processor architectures, as well as the support of loadable modules, which allows the process of adding a driver to be greatly simplified.

On many Linux-based embedded systems, displaying information has its own specifics. Often, a certain embedded system requires the development of specialized drivers for the applicable display controller.

When developing a display controller driver based on the Linux operating system, it is necessary to take into account a number of aspects related to the kernel development style, development patterns, and rapid changes in the kernel interaction interface (API). There are also few documents on the development of open-source display controller drivers, and are mainly limited to extracts from comments in the source code files of the Linux kernel, or schemes based on them. This article will look at a modern approach to writing a display controller driver for the Linux OS.

2. COMPONENTS OF THE GRAPHIC SYSTEM

Due to the need for 3D rendering, high-intensity calculation of shaders, high-resolution video processing and other requirements of a modern user of computer systems, the PC graphic subsystem has become significantly more complicated in comparison with it in the early years, and most often a modern graphic system includes [1]:

1. A Graphic accelerator (Graphics Processing Unit, GPU). It is its performance that is mainly measured by tests and various prototype software (for example, by startup of 3D applications at maximum settings or specially created scenes), it runs the shaders using a specialized multi-threaded architecture.
2. The display controller. This part of the graphics subsystem is used to determine the modes of graphics displaying, as well as to interact with monitors (obtaining available monitor resolutions, sending a finished frame, etc.).
3. You can also find specialized chips in the graphic system for video output, for interacting with video capture devices, and specialized random access memory (Graphics Double Data Rate, GDDR) for use by the graphic accelerator.

This list of components is not mandatory, since there are graphic subsystems in which one or more components are missing or not involved. For example, in some NVIDIA Optimus solutions installed in laptops, the on-screen display controller is not involved, the entire configuration of monitor modes occurs through the controller of another graphics card - Intel. Only a graphics accelerator is used from NVIDIA. You can also find systems with a different combination of selected components of the graphics system.

3. LINUX GRAPHIC SYSTEMS AND DRIVERS

The graphic system in Linux is represented by a graphic stack and various libraries used in it (**Fig. 1**) [2].

If we consider the components of the Linux graphical stack from top to bottom, we can say that the vast majority of them are designed to interact with the GPU, in particular: most of the Wayland and libdrm, fully EGL and almost completely X. Only a small part of DDX. (Device Dependent X) (Fig. 1.) and Wayland, as well as libgbm interacts with the on-screen display controller.

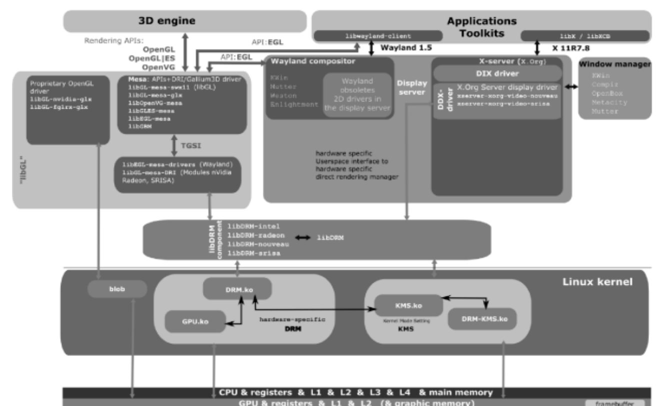

Fig. 1. *Graphic stack Linux.*

4.KERNEL MODE SETTING AND USER MODE SETTING. HISTORICAL BACKGROUND

Initially, there was only user mode setting, which was produced in the DDX drivers and enabled to set the mode for displaying graphical information. [3] The detemination was often executed through mode tables, and was not extensible, or it was done using algorithms within a code that were unique to each chip. There was also a solution such as VBE (Vesa BIOS Extensions), an extension for IBM PC BIOS, which allowed for some limited operations of installing modes with its use (without writing a direct algorithm for interacting with the graphics subsystem). Gradually, most drivers for IBM PCs switched to VBE. But VBE had some serious downsides:

- Lack of support for multiple CRT controllers (in most cases it is necessary to support output from multiple graphic accelerators to several displays or output to TV).
- Problems if there were several different outputs (and they could be different on the same chip series).
- Problems of use of several different timers.

To solve these problems in 2008, an extension was accepted to the kernel [4], which provided support for setting modes inside the kernel (kernel mode setting, or KMS). Along with the problems that this extension brought (such as some decrease in the stability of the kernel due to the fact that the drivers are written not very properly, as well as the expansion of the kernel because it includes many graphic drivers), there were also some advantages, such as simplified debugging (because there is the entire mode tuning code inside the kernel), simplified power management, and deleting the old code of the frame buffer drivers (in the era of tables and VBE (and for some drivers even now), the kernel also contained a frame buffer management subsystem, which allowed graphic output to be used without loading the graphic server for displaying console messages, or for drawing console programs).

The first version of the KMS protocol [5] was built on the XRandR 1.2 model, which was good enough at that time to install modes on ordinary workstations. But with the advent of mobile devices, the capabilities of the first KMS specification began to be insufficient - in order to reduce energy consumption, the concept of unified planes was developed, which was used to display video on a certain screen, to activate the hardware cursor (in the first versions of KMS, the hardware cursor interacted with the screen mode controller according to a separate protocol), and for other such cases.

Then support for other KMS objects was added, which allowed the gamma of the displayed output to be changed, the displayed image to be rotated directly in the driver, and so on [6].

There is one problem with all of this - for the interaction of these subsystems with the kernel, a large number of ioctl was required. And when trying to interact, each device had to check every ioctl, which was very inconvenient.

Due to these KMS problems on mobile devices, Google has developed its own interface to set modes, the Android Atomic Display Framework (ADF), for the Linux kernel as part of the Android OS [7].

Despite many advantages (atomic update of layers, ease of new drivers' development), ADF had the following disadvantages:

- There is only one update queue, and, therefore, the support of multi-screen modes is very poor. If there were 2 displays with different update rates, the update rate was led to the same (most often less) one, which resulted in either jerking at a faster display or slowing down at a slower one.
- ADF assumed that in user space there is a specific driver for the graphic chip, which necessarily provides 2D or 3D acceleration, which did not satisfy the developers of the main kernel, since it cut off all the "common" drivers (designed for devices not yet supported and providing minimal functionality) of the xf86-video-modesetting type.
- The ADF also had difficulties with updating the chain of outputs. Since drivers with support for multiple outputs have a lot of shared resources that require a one-time update regardless of the number of outputs, whereas the ADF had a requirement for an atomic update of the entire chain of one output. For multiple outputs, this could be implemented in a cycle, but this implementation did not work in the case of multiple outputs and shared resources.
- ADF used the midlayer pattern that is not recommended in kernel development. This pattern is not used, because despite the seeming usability and ease of developing drivers for the midlayer, there can always be a vendor that will be not satisfied with the midlayer - this is unacceptable for the kernel.
- ADF was new and there was no backward compatibility with old interfaces and APIs.

Therefore, it was decided not to save ADF, but to introduce its best features in KMS [5]. This is how KMS introduced the concept of atomic updates. The modern driver of the display controller must be written in the kernel space using this concept.

5. OBJECT MODEL OF THE LINUX KERNEL

To write a driver in kernel space, it is necessary to use its object model. [8] Despite the fact that the Linux kernel is written in C, there is an object model inside it that allows some object-oriented approaches to be applied when writing kernel drivers. To build this object model the following methods are most often used:

- Establishment of a mutually one-to-one correspondence between structures of a special type and objects (within the meaning of the OOP). The structure of a special type contains:
  - pointers to functions or a pointer to a vtable structure (where virtual methods are connected);
  - a pointer of the void* type, where the closed part of the structure data is connected (analogue of the closed type of inheritance);
  - contains the entire object of the base type (not in the form of a pointer) to ensure the connectivity between the parent and the heir through special compiler commands.
- Use of a special name and function signature (object_type_name_operation_name (object_type* object, ...)), which allows the type of object for which this function is intended to be precisely defined. It is possible to establish a one-to-one correspondence between methods in understanding the OOP and such functions.

- Vtable structures. C (unlike C++) does not have such an entity as virtual functions, tables of the virtual function, and similar things. Therefore, to store virtual methods, the developers of the kernel use structures consisting only of pointers to functions or storing in addition to such pointers also variables that are similar to the variables of the protected type of inheritance in OOP. These structures can be compared to a vtable from C++.
- To build private fields in some structures there are void* variables in which a closed structure can be assigned (in particular, when initializing the KMS driver, it is necessary to create a closed structure in the drm_device structure) - container_of together with storing the structure of the "base class" in the "heir". To simplify the implementation of inheritance in kernel structures, a method such as storing a complete basic structure within a derivative one is often used, and if it is necessary to obtain a derived structure from the basic one, the container_of macros is used, which internally uses the offsetof compiler command enabling to find the basic structure within the derivative one.

All of these things in general make writing code for the Linux kernel much easier and are used in most of the existing drivers.

6. KMS ATOMIC DROVERS. BASIC ENTITIES AND INTERFACES NECESSARY FOR IMPLEMENTATION

This section will consider the issue of creating a Linux kernel module that works through DRM and KMS with a display controller and does not use other components of the graphic subsystem. Such a driver is most often written for embedded systems, since components from different manufacturers (for example, a 3D chip from Mali, and on-screen display controller from Kirin) can be used in them, which must be considered when developing a driver for each component.

The main components of the KMS driver (**Fig. 2**) [9]:

1. Driver object (drm_driver). General parameters (such as its version, creator, and supported interfaces) are written in the driver structure body, and the general issues of driver operation (work with GEM, interrupts, and device files) are defined. As opposed to the general practice of writing objects in the kernel, the driver object contains most of the pointers to virtual methods (only file operations are added to the vtable structure). Initially, also work with vblank was determined there, but was transferred to CRTC (since in the case of multiple CRTC they may have different work with vblank events).
2. Controller of CRT (CRTC). The state of the mode displayed, objects of the main plane and cursor, the atomic state of the CRTC, and the activation bit are stored in CRTC. Functions assigned by CRTC

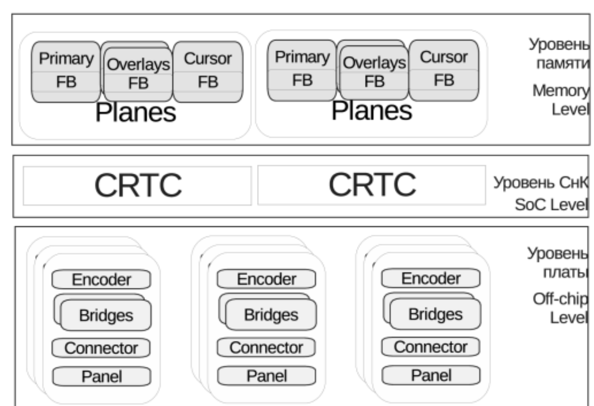

Fig. 2. *An example of the main components of the KMS driver.*

- working with vblank, with planes (main plane, cursor and overlay planes) and with pages.

3. One or several planes. For each CRTC there should be exactly one primary plane, no more than one cursor plane, and as many overlay planes as you want. Planes are attached to the CRTC that created them.
4. One or more encoders. The encoders must convert the signal from the internal DRM format to the output format and vice versa if a non-trivial conversion is required. It adjusts the mode, and also has functions for managing the power consumption of the information output device. The encoder is not rigidly linked to the CRTC, but instead it sets the bits possible to use by CRTC. The encoder may also check the set mode for admission, but usually it is used only in non-trivial cases.
5. Connector. The connector defines the output and connects the device directly and, through it, with one or more CRTC and encoders. Also, the connector has a status and it is there that the driver determines the mode check for whether it is possible to set it in this output. Most often the number of connectors corresponds to the number of video outputs. In many cases, the connector itself converts data into the necessary format, in this case it is linked to an empty encoder.
6. Bridge (drm_bridge). The bridge connects several encoders if the signal needs to be converted several times. In terms of functionality, it is similar to the connector and encoder.
7. Panel (drm_panel). The panel is a software representation of LCD panel control functions and is linked to the connector. It sets the functions for obtaining permissible modes and timings of the parent connector.

7. DEBUGGING DRIVER

To debug the kernel space driver [10], it is possible to use:

1. The kernel assembled in the Debug mode (and the interfaces of such a kernel, output via debugfs). Inside the kernel assembled supported by dynamic debugging, there is a debugfs virtual file system, in pseudo-files of which various debugging information is output on user request. The disadvantages of this system consist in the fact that it works only on the loaded kernel and it is necessary to read the files through the debugged PC, and this system also works only with the areas of the kernel code considered fit by the developers themselves. For arbitrary debugging it is necessary to make own debugging changes on the kernel.
2. The KGDB debugger. The KGDB debugger is GDB started up on a remote PC connected via a serial port interface to a debugged board. To do this, inside the kernel, there is an opportunity to assemble with support of waiting for a connection from the GDB interface, with a support of displaying error messages and stack tracing to the serial port. On the part of the remote user's PC, it is necessary to install the GDB debugger, as well as a kernel sample assembled in debug mode with debug information, only in this case KGDB will provide information about what is happening inside the kernel. To separate the regular output and the debugging output, special mixer programs working as an interlayer

between the debugger and the kernel are used.

3. Specialized FPGA (Field-Programmable Gate Array), such as Palladium, which allow graphics subsystem to trace the receipt of signals [11]. FPGA data has access to signals and values in the registers of devices that they emulate, which allows the value of kernel to be compared with the recorded values when debugging the drivers.

8. CONCLUSION

When developing an on-screen display controller driver for embedded systems, it is necessary to take into account a number of aspects, such as debugging difficulties, a special object model of the Linux kernel, the specifics of interaction with the hardware component. Since on-screen display controllers are available in almost each embedded system (except for those that do not use on-screen display at all), the problem of developing controller drivers for them, approaches to their design, debugging and testing in accordance with the Linux kernel object model is an relevant objective. Currently, the development of these drivers is carried out mainly in a closed way in large companies that make only the final result available to the public, which has not very positive impact on the development of a single approach to the development of on-screen display controller drivers and approaches to the methods of screen display modes setting in general.

This article considers the approaches to writing driver output controller modes used at different times in Linux OS, as well as reviews the latest approach to the development of data drivers for embedded systems in details. The practical implementation of the driver of such type can be used both with GPU drivers and with new developments in the field of software rasterization (in particular, llvmpipe).

Acknowledgment

Publication is made as part of national assignment for fundamental scientific researches (GP 14) on the topic (project) No.0065-2019-0001.

REFERENCES

1. Madieu J. *Linux Device Drivers Development.* Birmingham, Packt Publishing, 2017, 586 p.
2. Efremov IA, Mamrosenko KA, Reshetnikov VN. Metody razrabotki drajverov graficheskoj podsistemy [Methods of development graphics subsystem drivers]. Programmnye produkty i sistemy, 2018, 31(3):425-429 (in Russ.).
3. Verhaegen L. *X and Modesetting: Atrophy illustrated.* 2006, p. 7.
4. Airlie Dave, Barnes Jesse, Bornecrantz Jakob. DRM: add mode setting support [Electronic resource]. URL: *https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f453ba0460742ad027ae0c4c7d61e62817b3e7ef* (last access date: 04.09.2018).
5. Vetter Daniel. Atomic mode setting design overview, part 1 [Electronic resource]. URL: *https://lwn.net/Articles/653071/* (last access date: 04.09.2018).
6. Pinchart L. Why and How to use KMS as Your Userspace Display API of Choice. *LinuxCon Japan Tokyo*, 2013, 52 p.
7. Syrjälä V, Clark R, Hackmann G. [RFC 0/4] Atomic Display Framework [Electronic resource]. URL: *https://lists.freedesktop.org/archives/dri-devel/2013-August/044522.html* (last access date: 20.11.2018).
8. Brown Neil. Object-oriented design patterns in the kernel, part 1 [Electronic

resource]. URL: *https://lwn.net/Articles/444910/* (last access date: 04.09.2018).

9. The Linux Kernel Documentation [Electronic resource]. URL: *https://www.kernel.org/doc/html/v4.11/gpu/drm-kms.html* (last access date: 20.04.2018).
10. Bazhenov PS, Mamrosenko KA, Reshetnikov VN. Issledovanie i otladka komponentov dlya obrabotki trexmernoy grafiki operatsionnykh sistem [Research and debugging components for processing three dimensional graphics in operating systems]. *Programmnye produkty, sistemy i algoritmy*, 2018, 4:5 (in Russ.).
11. Bogdanov AYu. Opyt primeneniya platformy prototipirovaniya na PLIS "Protium" dlya verifikatsii mikroprotsessorov [Experience in applying the Protium SOC prototyping platform for verification of microprocessors]. Trudy NIISI RAN, 2017, 7(2):46-49 (in Russ.).