DOI: 10.15827/2311-6749.19.4.2

Display Transmitter Link Controller Design Technology for Linux OS

K.V. Pugin [1], *Programmer, rilian@niisi.ras.ru*
K.A. Mamrosenko [1], *Ph.D. (Engineering), Head of Department, mamrosenko_k@niisi.ras.ru*
V.N. Reshetnikov [1], *Dr.Sc. (Physics and Mathematics), Professor, Chief Researcher, rvn@niisi.ras.ru*

[1] *Center of visualization and satellite information technologies SRISA RAS, Moscow, Russia, 117218*

The main purpose of the paper is to develop a driver architecture for a display transmitter link controller when the controller has its own registers and memory. The developed architecture should reduce update changes in the implementation code and should not require any special tools or methods to develop its implementations. The paper analyzes publicly accessible DRM subsystem-based drivers and identifies two main architecture types, which serve as a basis for the majority of open source drivers. It also analyzes the strengths and weaknesses of the architecture types to achieve the above purpose. The identified architecture types were used to build a new architecture that has the strengths of both types, which allows achieving the purpose.

The paper also describes the developed driver debugging methods, which are based on the architecture under analysis and take into account the possibility of errors in the hardware, absence or insufficiency of controller documentation and incomplete emulation of the devices being developed. The results were evaluated during development of the DisplayPort driver for a perspective controller, and the driver was tested together with a prototype device and a monitor supporting the DisplayPort 1.1 standard.

The results of this paper can be used to create new transmitter link controller drivers for Unix-like systems both when a production state controller exists and when doing parallel development of a new controller and a driver for it.

Keywords: *Linux, driver, architecture, DisplayPort, embedded systems.*

Currently, graphic display units are developing at a high pace: new displays with a resolution over 1920x1080 are appearing, refresh rate and chrominance frequency are growing – some modern displays have a refresh rate of 120 HZ and more than 8 bit per a color channel already. This makes the use of new standards in development of display interface controllers allowing such use of the communication channel between a display and a display controller a relevant issue. Data transmission between a display and a display controller unit often requires a separate display transmitter link controller (DTLC) to be created, which will complement the display controller by converting the output in the form required by the display. To be able to transmit high-resolution video at a rate of 1 Gbit/s and higher, such a device requires complex program control (device-specific), as opposed to earlier protocols, which do not require such control, or where common control method standards were developed long ago. In the context of this article, such DTLCs are called complex DTLCs. As the number of tasks with the use of high-resolution video is growing, new approaches to the development of drivers for such (complex) devices are needed [1].

When DTLC drivers are developed in parallel with the development or upgrade of the device, the following problems are encountered:

1. Unsynced development of the driver and the controller causes delays in the development, as it becomes difficult to distinguish between errors attributable to the controller and those attributable to the driver.
2. Absence or insufficiency of documents in case of a partially closed source code cause errors in the development of the driver and the above described problem.
3. The necessity to adjust the driver architecture to the controller's features, as for various reasons, in some embedded controllers the standard can be implemented only partially.

To address these problems, several approaches, such as creation of a DSL (Domain Specific Language) [2], creation of a driver generator based on XML description [3] and use of verifiers [4], are employed. Such approaches have their strengths, but they also have weak points, the most important of which are the impossibility to use methods applied to partially closed source code of the controller and indirect costs related to development of auxiliary tools. This article offers a new driver architecture for parallel development (or upgrade) of the controller, which allows minimization of indirect costs in the process of implementation (does not require development of new DSL or new verifiers) and does not require a completely open source code of the controller. To implement the architecture, Linux OS was chosen as the most popular OS in embedded systems, while the development of modern drivers is most relevant for embedded systems. [5]

Research Methods

Approaches to Development of Display Controller Drivers Using Linux OS as an Example.

There are several known approaches related to the use of some known graphic subsystem tools that are employed in development of DTLC drivers for Linux. We will look at the most popular of them:

1. DRM (Kernel Mode Setting). The display transmitter link interface is a part of the Direct Rendering Management (DRM) infrastructure. Looking at the DRM subsystem as a software model of a real output device, all transmitter link interfaces refer to the connector type, as they connect a display and the display output controller, where the display is usually the connected element [6]. However, the connector type implementation cannot control power supply to the device and set the required mode, therefore complex DTLC drivers are most commonly implemented as a bridge + connector bunch, or much less often as an encoder + connector bunch. In contrast to the option with the use of a bridge, the second option does not refer to an external model, being though realizable in practice [6].
2. User Mode Setting (UMS) or an implementation with setting modes and interfacing with the hardware inside an X server. Problems in the implementation and employment of the UMS were identified in 2006 [7], after which drivers for transmitter link controllers were not actually developed using this approach.
3. FrameBuffer device (fbdev). This kernel interface preceded the DRM KMS unit and was most widespread till 2008-2009, when the first KMS version was taken into the kernel. The fbdev interface did not provide an implementation with setting up modes in the kernel, thus requiring the UMS implementation, which caused the above-mentioned problems, and after the KMS appeared, drivers for transmitter link controllers were not developed using this approach.
4. Driver implementations for Android Display Framework (ADF) are available, but they have the following weak points: standard hardware-independent functions of the protocol need to be developed from the ground up in each driver adding the possibility of errors and increasing the labor intensity. In addition, the ADF architecture (which is a development of the fbdev) causes the necessity to combine transmitter link controller drivers and display output controller drivers.

Implementations of controller drivers in some other Unix-compatible systems (e.g. FreeBSD) also use the DRM as the most developed open source system to implement complex display tools.

Employed DTLC Driver Analysis Methods.

The article involved an analysis of the existing open source DTLC drivers for Linux OS and combined drivers for video subsystems that include DTLC. In addition, a case study was conducted, which involved DisplayPort driver implementation for a prototype controller with the use of the analysis results. The case study included development of the DisplayPort driver architecture, which allowed quick switching from prototype control to standard device control. Then, the architecture developed in the course of the case study was extended to all complex DTLCs using the preceding analysis.

Case Study: DisplayPort

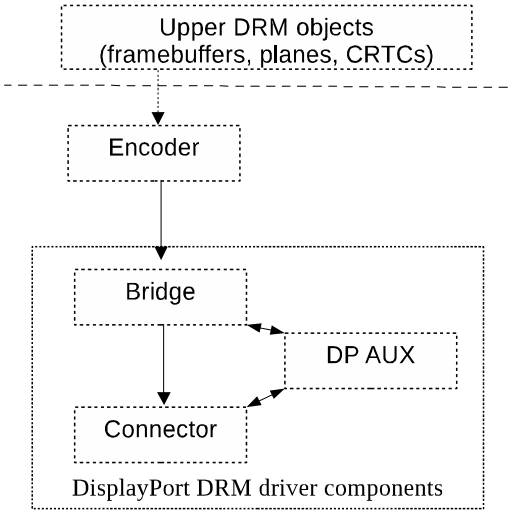

Fig. 1. DisplayPort software model in the KMS model hierarchy

One of the newest and most promising protocols for the display transmitter link is DisplayPort protocol, which was announced to be created at a conference in 2005 [8], with the first version adopted in 2006 [9]. The current version of the standard was adopted in 2016 (a new version was announced in 2019), and the USB-C connectivity offered by the protocol made it popular in embedded systems. The DisplayPort protocol is expected to become the most popular one for display link transmitting by 2019 [10].

Driver Architecture in the DRM Model.

To implement the DisplayPort protocol in DRM, the drm_connector and drm_bridge or drm_encoder types are used (to implement the on/off semantics and establish connection [11]. In this case, for reasons described in [6], and to ensure that the external DRM model is correct [6], the drm_bridge was chosen. As DisplayPort features not only interface with the display, but also additional functions (e.g. transmission via the DPCD and I2C protocols), DRM provides a special object DP AUX (drm_dp_aux) for its implementation and several functions designed to work with DisplayPort controllers only.

Initialization Sequence Implemented in the DRM Model.

In case of DRM, the typical work with DisplayPort is as follows:

1. Driver initialization – a drm_bridge (or drm_encoder) object is created with further creation of drm_connector and its connection to the previously created object.
2. drm_dp_aux initialization - a virtual device emulating the I2C slave bus is created and the additional channel is prepared for transmitting data to the display.
3. Polling of available displays via the HPD protocol (the HPD must be implemented in DisplayPort) using the standard detect function from the drm_connector API.
4. If the display is found, initialization and link training is performed. Various link training functions are implemented as standard functions of the DRM subsystem, but the main algorithm is implemented by the driver manufacturer in accordance with the standard. The link training is performed in two stages: clock recovery and channel equalization. After both stages are completed, video signal transmission is initialized.
5. When the video signal transmission is initialized, the mode of the display is received via the I2C protocol emulated using DPCD packages (Fig. 2).
6. When the video signal is transmitted, the channel is periodically tested for failures and errors. If such failures and errors are found, the initialization is partially or fully reset, and all the initialization functions, or a part thereof, are performed again for new display synchronization.
7. When the channel is tested, some devices send interrupts, which are processed by the driver initiating the actions of the DRM subsystem.

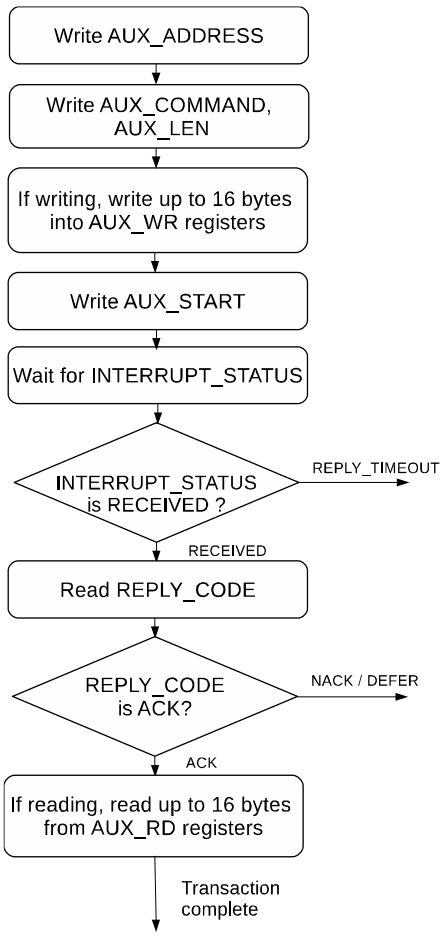

Fig. 2. Processing of messages from the auxiliary channel in the processing function for the device

To complete the initialization procedure successfully and provide the possibility of interfacing between the DisplayPort device and the display, information exchange procedures were implemented via the additional channel [11] The implementation can be performed either directly using packet messages, or through message exchange in the I2C bus format. To implement the procedures, a platform-specific function for direct packet exchange via DPCD is needed, which was implemented during creation of the prototype driver. The operation algorithm created for the device being developed is shown in Fig. 2. The DRM subsystem allows using the transmission function for I2C protocol operations, which is one of the benefits of the use of this subsystem in development of DisplayPort controller drivers.

Newer DisplayPort versions are also able to transmit sound via DisplayPort (this is relevant, for example, for display embedded speakers). As audio data are transmitted via the additional DisplayPort channel (similarly to I2C), the developed transmission function can be also used in this case. When audio data are transmitted, they are converted with the DRM tools.

Strengths and Weaknesses of the Current DisplayPort Implementation in the Linux Kernel.

When a proprietary DisplayPort driver was written for the DRM subsystem, the following strengths of the subsystem were found [6]:

1. An auxiliary channel object, which allowed to implement only device-specific actions (such as the device register configuration) without implementing the DisplayPort protocol parts associated with interfacing with the additional channels.
2. Capabilities provided in the drm_connector API to create nonstandard detection functions allowed implementation of the HPD required for correct DisplayPort operations in the DRM subsystem.
3. Additional adjustment functionality for mode setting and on/off control of the DisplayPort controller according to the external KMS model.

At the same time, the following weaknesses of the DRM subsystem were found for writing DisplayPort drivers:

1. No fully implemented standard procedure of clock recovery and channel equalization. The DRM subsystem has several functions facilitating the implementation of this procedure, but the main synchronization procedure is left to the discretion of the driver manufacturers, which can lead to errors in implementation of the protocol and errors in implementation of the procedure inside the driver, as well as different semantics of implementation of the procedure in various drivers.

2. No implemented mode setting in the drm_connector (not required in the external model). For the DisplayPort protocol, it is extremely important to set the correct mode of the controller and properly transmit it via DPCD to the display, while the drm_connector does not have such functions. At the current time, the majority of drivers use the drm_bridge to implement this functionality, as this object allows creation of drivers in line with the external model. Addition of the functionality to the drm_connector will allow a decrease in the time needed to write a driver and a reduction of the code base for drivers using the drm_bridge to implement mode setting only. For example, addition of the panel_bridge API allowed a reduction of the code base for the drivers using the drm_panel to implement operations with an embedded display [6].

Results and Discussion

Open Source Driver Architecture Analysis Results.

The analysis of the driver architectures described in [12] allowed identification of two typical display output subsystem drivers, which may be figuratively defined as 'all in one' (Fig. 3) and 'a specific driver for each hardware upgrade' (Fig. 4) The 'all in one' architecture is mainly used in drivers combined with GPU drivers, while for separate drivers the 2nd type of the architecture is typically used. Looking at the architecture shown in (Fig. 3), it can be noted that such a model does not allow the use of any DTLC drivers with any DC drivers; consequently, DC and DTLC drivers should be developed in parallel. However, for embedded systems, this cannot be always the case, as IP units from different manufacturers can be used for DC and DTLC. The approach allows identification of a common code, if the interface between instances is based on the linux kernel module instances interfacing mechanism (various instances of the same unit are used with various interfaces with hardware components) [13].

The 'a specific driver for each hardware upgrade' architecture (Fig. 4) requires a larger number of DTLC drivers to be developed, which is also a weakness, as it generates code duplication in case of similar controllers. Therefore, a driver architecture for embedded systems was developed, which allows a quick update of the interfacing unit with the hardware, and does not require any drivers to be developed for each DTLC controller version.

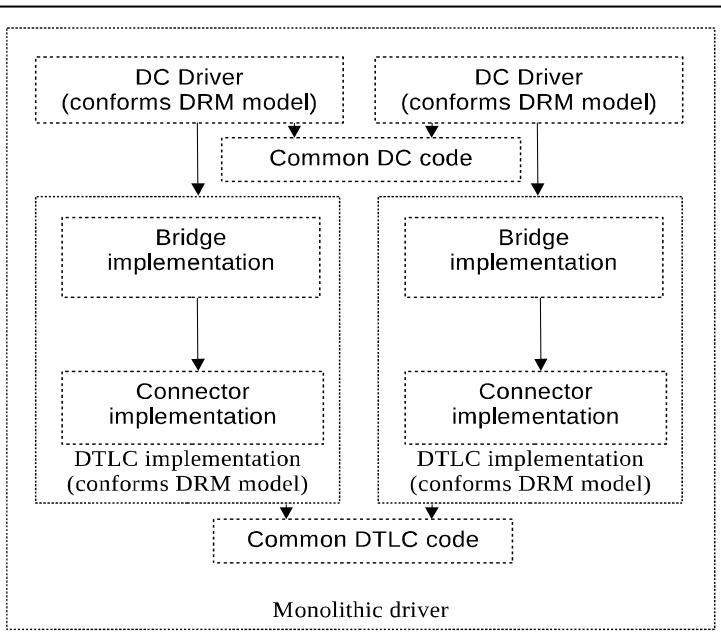

Fig. 3. Monolithic driver architecture for the display output subsystem

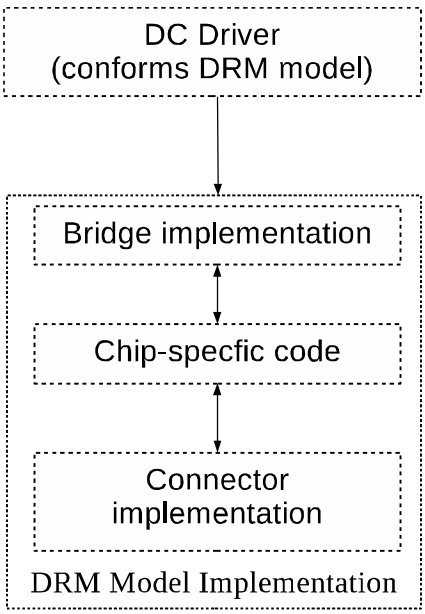

Fig. 4. DTLC driver architecture

DTLC Driver Architecture Requirements.

The case study and the analysis of the standards and documents allowed making the following findings:

1. Even if the hardware deviates from the standard, the DTLC driver architecture should be built using the available standard model and considering the specific features of the API provided by the OS (Fig. 1).
2. The functions of the hardware components extending the standard should be considered at the first stage only to the extent required for operational capability of the device (Fig. 2)
3. If the operation principles of the hardware deviate from the standard, the software components of the driver should be upgraded to support the required part of the standard in interfaces with the user space and display.

The ‘at least 1 driver for each device type’ principle should be followed in all cases. A combined driver has a lower update rate, and errors in one of its components interfacing with hardware can cause incorrect operation of all controlled devices resulting from being tightly coupled.

Architecture Developed for Embedded Systems.

For the DRM subsystem, the case study allowed circumventing the requirement to create a combined driver by using a bridge + connector, and the AUX component was shaped into a separate entity by creating an internal API, which allowed quick replacement of the unit interfacing with the hardware (in case of changes in the hardware), while the unit integrated with the operating system remained the same, thus allowing a high level of testing of the integration unit (Fig. 5)

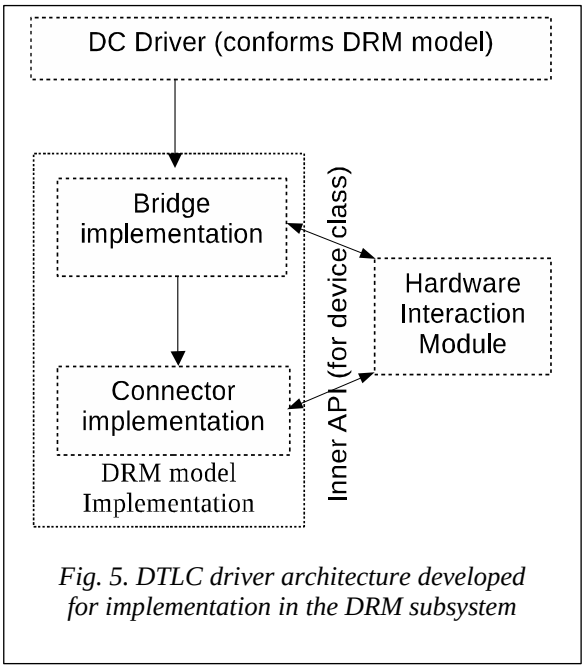

Fig. 5. DTLC driver architecture developed for implementation in the DRM subsystem

In comparison with the architectures of the known drivers supplied with the Linux OS kernel, the developed architecture possesses a unique combination of features:

1. Employment of drm_bridge in combination with the DeviceTree. According to the developer of the internal bridge structure, it is most applicable to embedded systems [14].
2. Employment of an operation principle, which is most close to the applicable DTLC standard. All known DTLC drivers that are not related to DC drivers [15] have deviations from the respective DTLC (HDMI or DP) standards.
3. In contrast to the known DTLC drivers based on the ‘all in one’ [16] or ‘a specific driver for each hardware upgrade’ [17,18] architectures, this architecture employs an internal API to optimize the support of several generations of IP units. An upgrade often involves addition of new functionality (e.g. switching to a new standard version) or correction of errors in hardware. In case of an internal API, the task to update a driver is limited to the update of its functions, as opposed to drivers based on other architectures.

Using a connection with ‘as closer to the hardware as possible and with separation of drivers using the DeviceTree’ (Fig. 4) and ‘a single combined driver, where the major part of the code is common’ (Fig. 3) architectures, we obtained a new DTLC driver architecture for embedded systems, which can be characterized as ‘the major part of the code is common, but communication via the DeviceTree’ (Fig. 5) and allows a reduction in the scope of changes in the implementation code, when the hardware is upgraded, and does not require any special tools to be developed.

Difficulties in Debugging of DTLC Drivers in Parallel with the Development of the Device.

Development of a driver can involve a situation, when apart from the software components of the driver, errors also appear in the hardware components of the transmitter link controller. The following means may be used for troubleshooting:

1. In case of absence or poor quality of documents, the protocol documents and the DRM subsystem specifications (as related to the selected protocol) should be additionally used. It should be borne in mind that prototype device documents can contain errors, therefore when data from such documents are used, they should be checked against the device emulation data.
2. When documents are available, data contained therein should be checked by direct reading and direct writing in the device registers, then the results obtained should be checked against the results delivered by the driver. When a kernel driver is developed, hardly traceable errors can appear, which lead to writing of incorrect values into the controller’s memory: such errors are much faster traced when the values are checked against those obtained by direct writing.
3. When bare-metal tests are used for the device, the values they write into the controller’s memory should be traced and used to check the driver’s operation in the same mode, which will allow distinguishing errors in the hardware components (in this case, the test is performed improperly) and the driver (the test is performed properly).
4. Earlier prototype devices can implement incomplete functionality described in the documents and the standard: for example, pixel frequency control can be missing. This imposes some limitations on the tests, which should by circumvented by testing the developed controller in marginal conditions and more applicable conditions.

If the device reveals incorrect operation while the driver is operating in according with the documents and the bare-metal test data, in can be assumed that the hardware components of the device have problems which should be corrected either using the driver or by bare-metal testing of the prototyping system [19]. To test the developed

method, a driver was developed, which was tested in parallel with emulation of the prototype device using the Altera system with a display supporting the DisplayPort 1.1a being used as an output device.

Conclusion

This article offers a new DTLC driver architecture based on the analysis of the existing Linux OS DRM subsystem-based drivers, which architecture allows optimization of the development and upgrade of the drivers.

The area of further research into the architectures of graphic subsystem drivers will involve an analysis of drivers for embedded systems with an external bus (e.g. employment of DTLC in PCI-E) and new case studies in respect of specific DTLC protocols (via HDMI).

Acknowledgments: The publication is made as a part of a national assignment for SRISA RAS (fundamental scientific research GP 14) on the topic no. 0065-2019-0001 (AAAA-A19-119011790077-1).

References

Corbet J., Rubini A., Kroah-Hartman G. *Linux Device Drivers*. O'Reilly, 2005, 615 p.

2. Lisboa E.B., et al. An approach to concurrent development of device drivers and device controller. *Proc. 11 ICACT*, Phoenix Park, Korea (South), IEEE, 2009, pp. 571–575.

3. Jung Choon Park, Yong Hoon Choi, Tae ho Kim. Domain Specific Code Generation for Linux Device Driver. *Proc. 10 ICACT*, Gangwon-Do, Korea (South), IEEE, 2008, pp. 101–104. DOI: 10.1109/ICACT.2008.4493721.

4. Dileep K.P., et al. Rules Based Automatic Linux Device Driver Verifier and Code Assistance. *Proc. ICRAIE*, Jaipur, India, IEEE, 2014. DOI: 10.1109/ICRAIE.2014.6909321.

5. Oluwole Oyetoke. Embedded Systems Engineering, the Future of Our Technology World; A Look into the Design of Optimized Energy Metering Devices. *IJRES*, 2015, vol. 18, pp. 17–21.

6. *Linux GPU Driver Developer's Guide*. 2019. Avaible at: https://dri.freedesktop.org/docs/drm/gpu/index.html (accessed Jule 06, 2019).

7. Verhaegen L. *X and Modesetting: Atrophy illustrated*. 2006. Avaible at: https://people.freedesktop.org/~libv/X_and_Modesetting_-_Atrophy_illustrated_%28paper%29.pdf (accessed Jule 06, 2019).

8. Kobayashi A. DisplayPort technical overview. *Proc. 25th Int. Display Research Conf. EURODISPLAY 2005*. Edinburgh, Scotland, UK, 2005, pp. 98–101.

9. *VESA DisplayPort Standard Version 1. Video Electronics Standards Association*, 2006. Avaible at: https://glenwing.github.io/docs/DP-1.0.pdf (accessed Jule 06, 2019).

10. Noman Akhtar, Dinesh Kithany. *DisplayPort expected to surpass HDMI in 2019*. 2018. Avaible at: https://technology.ihs.com/599141/displayport-expected-to-surpass-hdmi-in-2019 (accessed Jule 06, 2019).

11. *VESA DisplayPort Standard Version 1, Revision 1a. Video Electronics Standards Association*, 2008. Avaible at: https://glenwing.github.io/docs/DP-1.1a.pdf (accessed Jule 06, 2019)

12. *DRM Subsystem drivers. 2019.* Avaible at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm (accessed Jule 06, 2019).

13. Bovet D.P., Cesati M. *Understanding the Linux Kernel*. O'Reilly, 2005, 942 p.

14. Sean P. *DRM: Add drm_bridge*. 2013. Avaible at: https://lwn.net/Articles/563156/ (accessed Jule 06, 2019).

15. Hajda A., et al. *DRM Bridge drivers.* 2019. Avaible at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/bridge (accessed Jule 06, 2019).

16. Nikula J., Lahtinen J., Vivi R. *Intel DRM Driver.* 2019. Avaible at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/i915/ (accessed Jule 06, 2019).

17. Hajda A., Purski M. *TC 358764 DRM Driver*. 2019. Avaible at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/bridge/tc358764.c. (accessed Jule 06, 2019).

18. Gusakov A., Zabel Ph. *TC 358767 DRM Driver*. 2019. Avaible at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/bridge/tc358767.c. (accessed Jule 06, 2019).

19. Bogdanov A.Y. *Experience in the Use of Protium FPGA-Based Prototyping Platform to Verify Microprocessors*. Proc. of the SRISA RAS, 2017, vol. 7, no. 2, pp. 46–49.

УДК 004.454
DOI: 10.15827/2311-6749.19.4.2

Технология разработки драйвера контроллера сопряжения с устройством отображения информации для системы Linux

К.В. Пугин [1], *программист, rilian@niisi.ras.ru*
К.А. Мамросенко [1], *к.т.н., доцент кафедры космических телекоммуникаций, заведующий отделом, mamrosenko_k@niisi.ras.ru*
В.Н. Решетников [1], *д.ф-м.н., профессор, главный научный сотрудник, rvn@niisi.ras.ru*

[1]*Центр визуализации и спутниковых информационных технологий НИИСИ РАН, г. Москва, 117218, Россия*

Главная цель статьи – разработка архитектуры драйвера для контроллера сопряжения с устройством отображения информации для случаев, когда у контроллера есть свои собственные регистры и память. Разработанная архитектура должна уменьшить количество обновлений кода реализации и не должна требовать каких-либо специальных инструментов или методов для разработки своих реализаций. В статье анализируются общедоступные драйверы на основе подсистемы DRM и определяются два основных типа архитектуры, которые служат основой для большинства драйверов с открытым исходным кодом. Далее анализируются сильные и слабые стороны типов архитектуры на предмет возможности достижения вышеуказанной цели. Выявленные типы архитектуры были использованы для построения новой архитектуры, которая имеет сильные стороны обоих типов, что позволяет достичь цели.

В статье также описываются разработанные методы отладки драйверов, основанные на анализируемой архитектуре и учитывающие возможность ошибок в оборудовании, отсутствие или недостаточность документации контроллера и неполную эмуляцию разрабатываемых устройств. Результаты оценивались во время разработки драйвера DisplayPort для перспективного контроллера, и этот драйвер был протестирован вместе с прототипом устройства и монитором, поддерживающим стандарт DisplayPort 1.1.

Результаты работы могут быть использованы для создания новых драйверов контроллера сопряжения с устройством отображения информации для Unix-подобных систем, как при наличии контроллера состояния производства, так и при параллельной разработке нового контроллера и драйвера для него.

Ключевые слова: *Linux, драйвер, архитектура, DisplayPort, встроенные системы*.

Благодарности: *исследование проведено в рамках государственного задания ФГУ ФНЦ НИИСИ РАН (выполнение фундаментальных научных исследований ГП 14) по теме № 0065-2019-0001 "Математическое обеспечение и инструментальные средства для моделирования, проектирования и разработки эле-ментов сложных технических систем, программных комплексов и телекоммуникационных сетей в различных проблемно-ориентированных областях" (АААА-А19-119011790077-1).*

Литература

1. Corbet J., Rubini A., Kroah-Hartman G. Linux Device Drivers. O'Reilly, 2005, 615 p.
2. Lisboa E.B., et al. An approach to concurrent development of device drivers and device controller. Proc. 11 ICACT, Phoenix Park, Korea (South), IEEE, 2009, pp. 571–575.
3. Jung Choon Park, Yong Hoon Choi, Tae ho Kim. Domain Specific Code Generation for Linux Device Driver. Proc. 10 ICACT, Gangwon-Do, Korea (South), IEEE, 2008, pp. 101–104. DOI: 10.1109/ICACT.2008.4493721.
4. Dileep K.P., et al. Rules Based Automatic Linux Device Driver Verifier and Code Assistance. Proc. ICRAIE, Jaipur, India, IEEE, 2014. DOI: 10.1109/ICRAIE.2014.6909321.
5. Oluwole Oyetoke. Embedded Systems Engineering, the Future of Our Technology World; A Look into the Design of Optimized Energy Metering Devices. IJRES, 2015, vol. 18, pp. 17–21.
6. Linux GPU Driver Developer's Guide. 2019. URL: https://dri.freedesktop.org/docs/drm/gpu/index.html (дата обращения: 06 июля 2019).
7. Verhaegen L. X and Modesetting: Atrophy illustrated. 2006. URL: https://people.freedesktop.org/~libv/X_and_Modesetting_-_Atrophy_illustrated_%28paper%29.pdf (дата обращения: 06 июля 2019).
8. Kobayashi A. DisplayPort technical overview. Proc. 25th Int. Display Research Conf. EURODISPLAY 2005. Edinburgh, Scotland, UK, 2005, pp. 98–101.
9. VESA DisplayPort Standard Version 1. Video Electronics Standards Association, 2006. URL: https://glenwing.github.io/docs/DP-1.0.pdf (дата обращения: 06 июля 2019).
10. Noman Akhtar, Dinesh Kithany. DisplayPort expected to surpass HDMI in 2019. 2018. URL: https://technology.ihs.com/599141/displayport-expected-to-surpass-hdmi-in-2019 (дата обращения: 06 июля 2019).

11. VESA DisplayPort Standard Version 1, Revision 1a. Video Electronics Standards Association, 2008. URL: https://glenwing.github.io/docs/DP-1.1a.pdf (дата обращения: 06 июля 2019).

12. DRM Subsystem drivers. 2019. Avaible at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm (дата обращения: 06 июля 2019).

13. Bovet D.P., Cesati M. Understanding the Linux Kernel. 3rd ed. O'Reilly, 2005.

14. Sean P. DRM: Add drm_bridge. 2013. URL: https://lwn.net/Articles/563156/ (дата обращения: 06 июля 2019).

15. Hajda A. et al. DRM Bridge drivers. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/bridge (дата обращения: 06 июля 2019).

16. Nikula J., Lahtinen J., Vivi R. Intel DRM Driver. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/i915/ (дата обращения: 06 июля 2019).

17. Hajda A., Purski M. TC 358764 DRM Driver. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/bridge/tc358764.c. (дата обращения: 06 июля 2019).

18. Gusakov A., Zabel Ph. TC 358767 DRM Driver. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/bridge/tc358767.c. (дата обращения: 06 июля 2019).

19. Богданов А.И. Опыт применения платформы прототипирования на ПЛИС "Protium" для верификации микропроцессоров // Труды НИИСИ РАН. 2017. Т. 7. № 2. С. 46–49.