DOI: 10.17725/rensit.2023.15.453

Flat Panel Display Controller Driver Architecture for Linux OS

Konstantin V. Pugin, Kirill Mamrosenko, Alexander Giatsintov
Scientific Research Institute of System Analysis of RAS, https://niisi.ru/
Moscow 117218, Russian Federation
E-mail: rilian@niisi.ras.ru, mamrosenko_k@niisi.ras.ru, algts@niisi.ras.ru

Received May 25, 2023, peer-reviewed June 01, 2023, accepted June 08, 2023, published December 06, 2023.

Abstract: **This paper discusses the development of a driver architecture for display transmitter link controller. The architecture ensures the implementation of protocols for interaction with flat panel displays in the case when the controller has its own registers and configuration system. Unlike the known solutions, the proposed architecture makes it possible to reduce the amount of changes in the implementation code in the event of hardware upgrade, and also does not require the use of automatic driver code generation based on high-level descriptions or the development of special tools, such as domain-specific languages. This paper analyses drivers that are based on Direct Rendering Management subsystem and available in open source, as well as previously described approaches to the development of display transmitter link controller drivers. The paper also presents a logical comparator model for testing phase-locked loop devices, which are an integral part of all display output stacks. Based on this model, an IP block was developed, which was used to test the Display Serial Interface driver. Evaluation of the results was carried out in the development of the MIPI Display Serial Interface driver for a promising controller. This driver was tested together with a device prototype and a panel that supports the MIPI Display Serial Interface 1.3 standard. The results provided in this paper can be used both to develop new drivers for existing controllers and new controllers with new drivers.**

Keywords: **driver, architecture, MIPI DSI, embedded systems**

UDC 004.454

Acknowledgments: The work was carried out within the framework of the state task of the Federal State Institution FNTs NIISI RAS "Conducting fundamental scientific research (47 GP)" on topic No. FNEF-2022-0022 "Mathematical support and tools for modeling, designing and developing elements of complex technical systems, software systems and telecommunication networks in various problem-oriented areas".

For citation: Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov. Flat Panel Display Controller Driver Architecture for Linux OS. *RENSIT: Radioelectronics. Nanosystems. Information Technologies*, 2023, 15(4):453-462e. DOI: 10.17725/rensit.2023.15.453.

Contents

1. INTRODUCTION

Currently, graphic display units for mobile devices are developing at a high pace. Many new display devices are characterized by resolutions up to 2560x1600 and dynamic refresh rates from 60 to 120 Hz. In this regard, it becomes relevant to use new standards in developing display interface controllers that can stream video at high speed. To solve this problem, there are several existing protocols that require using separate devices, so called display transmitter link controllers (here and afterwards referred to as DTLC), which complement the display controller by converting the output in the form required by the display. Many protocols that transmit video at high speed (1 Gbit/s and higher) require both to interact dynamically and agree on acceptable transmission parameters. Controllers for such protocols require complex program control that is specific for every type of device. Protocols for flat panel displays that transfer smaller amounts of data often do not require such complex control and feedback. If these protocols are used, flat panel display configuration does not change when display is up and running, and dynamic configuration is not supported. Like [1], let's call complex DTLC a DTLC that require program control with feedback. As the number of mobile devices with displays that require new ways of interacting and new controllers increases, it is necessary to have approaches at hand to develop drivers for such (complex) devices [2]. If the DTLC driver for interacting with flat panel displays is developed in parallel with the device, then several problems are encountered, some of which are identical to those described in [1], and some are unique, related to the essence of these protocols:

1. There is a need to compute compatible frequencies for the FPGA and select flat panels that will be used during the design of a controller.
2. Possible incompatibility of allowed frequencies of panels and controllers, to bypass which additional hardware and software is required.
3. The need to test several frequencies and panel operation modes with no possibility to replace device hardware.
4. The need to bypass OS interfaces if they are not compatible with the protocol version that is implemented in hardware.

To address these problems, different hardware and software-based approaches are used: use of verifiers [3], creation of the Domain Specific Languages (DSLs) [4], automatic generation of driver code based on the common pattern [5], and development of drivers that define the unchangeable main features of a series of devices until the end of its production, while other features are parametristic and dynamically fetched from configuration file entries. This paper is an attempt to answer what approaches to the development of the DTLC driver for interacting with flat panel displays should be applied, given the proposed conditions. The contribution of the study is as follows:

- This paper improves the DTLC driver architecture created earlier by ([1]) for cases where DTLC is used to interact with the flat panel display.
- To test the correctness of programming an interface of phase-locked loop driver (afterwards referred to as PLL), at the early stages of the development of the IP block, the logical comparator model was created and implemented on FPGA, which is used in conjunction with built-in PLLs. This model made it possible to test the correctness of programming of phase-locked loop devices for DTLC in FPGA states similar to the real device application case.

The developed architecture ensures the implementation of protocols for interaction with flat panel displays in the case when the controller has its own registers and configuration system. Unlike the known solutions, the proposed architecture makes it possible to reduce the amount of changes in the implementation code in the event of hardware upgrade, and also does not require the use of automatic driver code generation based on high-level descriptions or the development of special tools, such as domain-specific languages.

Section 3.1 describes approaches to developing drivers for Linux OS, on the basis of which the study was carried out. Section 3.3 describes the architecture of the DTLC driver for flat panel displays, which contains two loosely bound components – a hard-ware binding component and an OS interaction component, which are linked by an internal API. The hardware binding component performs most of the necessary conversions of incoming data into those formats that are required for programming of a particular device, programs its registers and ensures feedback. The OS interaction component converts requests from other parts of the system and transmits them with an internal API, which is a system of functions and structures common to all hardware binding modules. Section 4 describes how to use the above mentioned architecture when developing a driver for the MIPI Display Serial Interface (MIPI DSI), which is one of the DTLC protocols for interacting with flat panel displays. Section 5 describes the logical model of the comparator for testing the driver using FPGA PLL generator. This model allows testing the correctness of programming the real hardware PLL drivers when it is necessary by using emulation of real hardware PLL interface in FPGA instruments.

2. RELATED WORK

Some recent papers [6] offer to create systems with several DTLCs for interacting with the flat panel display that have a common physical interface layer (afterwards referred to as PHY), which can be used by only one DTLC at a time. One may try to get round this limitation using the DTLC driver architecture described in [7]. In this architecture frequency synthsizer in the shared device, but in the architecture, described in [6], PHY will be the shared device in this architecture and not the frequency synthesiser. Or it is possible to preserve the limitation on simultaneous operations but separate out the frequency synthesiser (FS) together with PHY into an independent module with a common part of the driver. Since PHY described in [6] is used in DTLC to interact with the flat panel display, the driver architecture described in this paper can also be used for controllers with such a PHY (taking into account the paper by [7]).

The article [8] describes approaches to designing DTLC drivers for real-time operating systems. The description of the RTOS subsystem provided in this article allows applying the models for driver development like those used for DRM in Linux with minimal modification in terms of component names replacement. This allows developing DRM-compatible models for DTLC drivers to interact with the flat panel display not only for Linux but also for other OS, which is also reported in [9]. This paper cites that DRM is also used for writing graphics drivers for FreeBSD OS. So it is possible to use the models developed for DRM to write DTLC drivers in these OS.

3. RESEARCH APPROACHES

3.1 Approaches to the development of graphics controller drivers on the example of Linux OS

Several approaches to the use of certain popular graphics subsystem tools used in the development of DTLC drivers for Linux are known. The most popular of them are the following:

1. DRM. As mentioned in [10], the DTLC interface is part of the Direct Rendering Management (DRM) framework. If to consider the DRM subsystem as a software model of the real output device, then all flat panel displays and interfaces for their interaction with DTLC match the panel type because they determine how graphics output works with fixed frequencies. However, the panel type implementation requires that each type of panel has its own driver, even if the controllers match. Therefore, most often DTLC drivers for fixed panels are implemented as a bridge + panel bundle, or, less often as encoder + panel bundle. The second option, in contrast to the first one, does not correlate with the external model, although it is practically possible [11].
2. User Mode Setting (UMS) or the implementation of setting modes and interacting with hardware inside the X server. Problems with the implementation and the use of UMS were reported in 2006 [12], whereafter no drivers for DTLC were developed using this approach.
3. FrameBuffer device (fbdev). This kernel interface preceded the KMS DRM module and was most common until 2008-2009, when the first version of KMS was adopted. The fbdev interface did not provide for the implementation of mode configuration in the kernel, and, therefore, required the implementation of UMS, which led to the above problems. No drivers, specifically for DTLC devices, were developed using this approach after the appearance of KMS.
4. There are driver implementations for the Android Display Framework (ADF). However, they are standard, hardware-independent, protocol functions must be developed from scratch in each driver, which increases the likelihood

of errors and increases the complexity of the task. Also, due to the use of the ADF architecture (the development of fbdev), it becomes necessary to combine DTLC drivers and display controller drivers. Implementations of controller drivers in a number of other Unix-compatible systems (for example, FreeBSD) are also created with DRM ([9]), as the most advanced open system for implementing complex output tools.

3.2 Methods used to study DTLC drivers

When preparing the paper, the authors analysed the existing open DTLC drivers for Linux OS. The special emphasis was placed on drivers for mobile platforms. Based on this analysis, the novel Display Serial Interface (DSI) driver architecture was developed. It allows quickly creating the driver for a device prototype and, after hardware release, port this driver to a serial device. Through previous analysis and case study, and using materials from [1], the proposed architecture has been further extended to all complex DTLC drivers to interact with flat panel displays.

3.3 Differences in the DTLC driver architecture for interacting with the flat panel display

For DTLC with a dynamic panel, the architecture offered by [1], **Fig. 1**, is currently the most applicable one. To adapt this architecture for use in flat panel display systems, the drm_panel structure must be used, which will only work with DRM model interfaces, and the panel driver must be fully compatible with the DRM model. The improved architecture is shown in **Fig. 2**. In this version, the drive rarchitecture also retains all the advantages

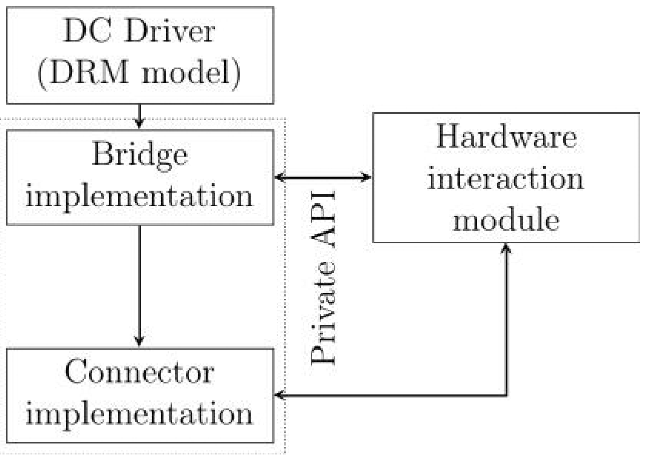

Fig. 1. *DTLC driver architecture for implementation within the DRM subsystem.*

described in [1], while being able to work with flat panel displays in cases where the feedback implementation in the panel complies with the relevant protocol clauses. However, this architecture needs to be improved in cases where non-standard extensions are required to interact with the panel interface.

4. ANALYSIS OF THE MIPI DISPLAY SERIAL INTERFACE PROTOCOL

Along with eDP (embedded DisplayPort, the development of drivers for which was described in [1]), MIPI DSI, used by leading Android mobile device manufacturers, is one of the most used protocols for transmitting data

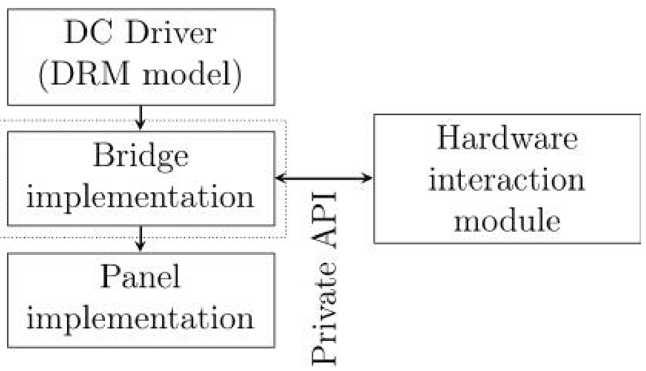

Fig. 2. *DTLC driver architecture for interacting with flat panel displays for implementation within the DRM subsystem.*

to displays. The important part of this protocol is a feedback protocol that is implemented through the Display Command Set (DCS), which defines the format of commands and responses of interaction participants, as well as a list of standard commands that must be supported by all compatible devices. To implement the MIPI DSI protocol, DRM uses drm_encoder (to implement the controller driver), drm_panel (to implement the panel driver), and drm_connector or drm_bridge (to implement panel interaction with the controller) [11].

There is a program model in the Linux kernel for DRM-based MIPI DSI drivers that complements the general DRM model as follows: MIPI DSI, like similar eDP standard implements not only the basic image output, but also additional functions (for example, the transfer of various data via the DCS protocol). To implement these functions from the MIPI DSI controller side, the DRM has a MIPI DSI Host object (mipi_dsi_host), and to implement these functions from the panel side, there is the MIPI DSI Device object (mipi_dsi_device), as well as several

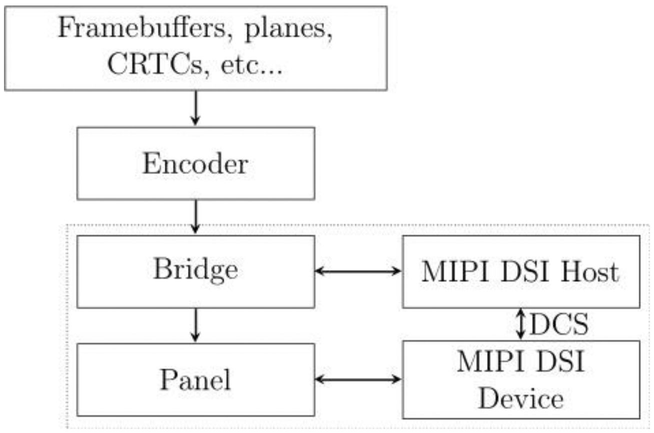

Fig. 3. *MIPI DSI program model within the common DRM model.*

auxiliary functions only for MIPI DSI controllers and panels. MIPI DSI DRM model is shown on **Fig. 3**.

4.1 Peculiarities of working with MIPI DSI displays

The vast majority of PHY for MIPI DSI (afterwards referred to as D-PHY) [13] either use burst mode or require certain constants for the MIPI DSI protocol to be taken into account. This leads to the fact that, unlike other DTLC, in MIPI DSI it is necessary to recalculate the time and frequency characteristics for all the modes. Also, unlike the DisplayPort protocol, the data transfer frequency in DSI is not constant but is derived from the pixel frequency using the formula

$clk_{hs} = clk_{pix} \cdot BPP \cdot lanes,$

where lanes is the number of active transmission lines (from 1 to 4), BPP is the number of bits per pixel, and clkpix is pixel clock. It is the bit frequency that must be transmitted to the PLL of the DSI controller [14]. The DSI protocol uses the Display Monitor Timings (DMT, see [15]) definition of screen resolution. However, unlike other protocols, DSI needs to recalculate all the characteristics of horizontal lines in order for the invisible part of the screen to fit the packet headers. Let us denote the horizontal characteristics as: HBP – Front Porch, HFP – Back Porch, HSA – Hsync Active, HACT – H Active, PULSE_CLK – pulse synchronisation value, – число битов в пикселе. number of bits in a pixel. For each characteristic (denoted as X) let’s XDPI be input characteristic of X, XDSI be recalculated characteristic of X. Protocol constants, such as HDR = 6 (packet header, one per each timing, must be in the invisible area), HSS = 4 horizontal synchronisation start header), HSE = 4 (horizontal synchronisation end header), are also important for recalculation. The following formulas are used to recalculate timings:

$$HBP_{DSI} = (HBP_{DPI} \cdot BPP/8) - HDR_{HBP}$$
$$HFP_{DSI} = (HFP_{DPI} \cdot BPP/8) - HDR_{HACT} - HDR_{HFP}$$
$$HSA_{DSI} = (HSA_{DPI} \cdot BPP/8) - HSS - HDR_{HSA} - HSE$$
$$HACT_{DSI} = HACT_{DPI} \cdot BPP/8$$
$$PULSE\_CLK = ((HACT_{DPI} + HSA_{DPI} + HBP_{DPI} + HFP_{DSI}) \cdot BPP/8) - HSA_{DSI} - 20.$$

Calculations in accordance with the above formulas must always deliver integers. If the figure turns out to be non-integer, then the driver must choose an alternative. Since there is no resources for this in the common DRM stack, this should be redirected to the controller driver. Different controller models may have different additional restrictions, such as the inability to select certain bit rates or the need to select specific vertical timings. To be able to determine additional restrictions on controller values, a module originally proposed in [1] for hardware interaction is used. However, an improvement is required for use with the fixed panel (2). It is also possible to reduce Vertical Front Porch (VFP, see [15]) to reduce power consumption with high-speed operation, which is performed depending on the model of the controller and panel. Therefore, work on reducing VFP is done in the hardware module interactions. To test the correctness of programming the transmission frequency generator, a testing comparator (5) was used, which made it possible to avoid PLL programming errors when testing the DTLC driver for the MIPI DSI controller.

5. COMPARATOR-BASED LOGIC MODEL FOR TESTING PLL DRIVERS

For most DTLC, the driver needs to correctly program PLL frequencies directly related to the image output. On some controllers, the PLL data programming interface can be built directly into the controller's program interface and performed in the same way as all other setup operations. Despite the development of phase-locked loop devices with reduced size [16], FPGA and emulators often lack the ability to dynamically tune PLL, which makes it very difficult to test DTLC controllers with changing the connected monitor without powering off the controller, for example, DisplayPort or HDMI. Each frequency value required project rewiring for FPGA. For DTLC with fixed panels, this problem is less relevant, since panels support a small frequency range and, thereore, a small number of supported display modes. Such systems have another problem with FPGA drivers. Often, the programmable value of the PLL drivers is not taken into account by FPGA, which allows FPGA-based controller prototypes to display an image with incorrectly programmed PLL. In this case, the fact of incorrect programming would manifest on production devices. To solve this problem, it was proposed to develop a testing block for FPGA-based implementation, the logical model of which consists of the following elements (**Fig. 4**):

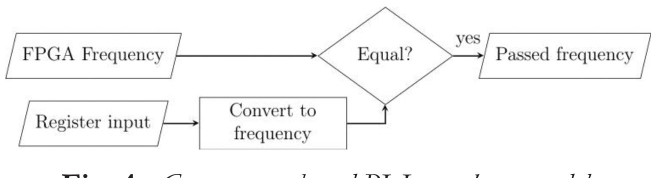

Fig. 4 . *Comparator-based PLL mock-up model.*

]

- Input that receives the frequency specified during FPGA wiring.
- Calculation block that converts the input to a frequency value (rounded to the nearest whole number). The exact structure of the calculation block is determined by the PLL programming interface or the part of the controller programming interface that programs PLL.
- Comparator that compares the calculation block output with the frequency specified during FPGA wiring. If the comparison result is positive, the comparator sends the enabling signal to the frequency skip switch. The allowed deviation of the estimated frequency from the incoming one depends on the device under development (it often does not exceed 1 kHz for DTLC).
- Frequency skip switch, which is turned on by the enabling signal from the comparator.

This model was implemented as an IP block on FPGA. The use of this model to test DTLC drivers for interaction with the flat panel display made it possible to achieve correct PLL programming for the DTLC controller even before the first prototype was made. The correctness of programming has been confirmed already at the testing stage with the use of emulators and FPGA.

6. CONCLUSION

The approaches proposed in this paper make it possible to reduce the amount of changes in the implementation code in case of hardware upgrade. Also, these approaches make it possible to reduce the number of driver modifications to support device families (including for one device with differences in various blocks, for example, different PLL). The PLL model described above, when implemented, will allow testing drivers at the early stages of PLL development, as well as device drivers that use PLL.

Further developments may deal with determining the applicability of the resulting architecture to the development of DTLC drivers for embedded systems with external buses (for example, using DTLC on PCI Express), researching specific DTLC protocols (over HDMI), and also determining the possibilities of using the comparator to test PLL drivers, used in other SoC parts.

REFERENCES

1. Pugin KV, Mamrosenko KA, Reshetnikov VN. Display Transmitter Link Controller Design Technology for Linux OS. *Software Journal:Theory and Applications*, 2019, 4:10-17, doi: 10.15827/2311-6749.33.406.
2. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 2005. ISBN: 0-596-00590-3.
3. Dileep KP, Raghavendra A, Suman M, Devesh G, Srikanth SV. Rules Based Automatic Linux Device Driver Verifier and Code Assistance. *Proc. IEEE International Conference on Recent Advances and Innovations in Engineering*. Jaipur, India: IEEE, 2014. ISBN: 978-1-4799-4040-0.
4. Lisboa EB, Silva L, Lima T, Chaves I, Barros E. An Approach to Concurrent Development of Device Drivers

and Device Controller. *Proc. 11th International Conference on Advanced Communication Technology, pp.* 571-575. Phoenix Park: IEEE, 2009. ISBN: 978-89-5519-139-4.

5. Jung Choon Park, Yong Hoon Choi, Tae Ho Kim. Domain Specific Code Generation For Linux Device Driver. *Proc. 10th International Conference Advanced Communication Technology, pp.* 101-104. Gangwon-Do, Korea (South): IEEE, Feb, 2008. ISBN: 978-89-5519-136-3. DOI: 10.1109/ICACT.2008.4493721.12.
6. Sunil Kumar CR, Aruna Kumar, Sanjib Basu. Novel Circuit Architecture for Configurable eDP and MIPI DPHY IO. *Proc. 2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems* (*VLSID*), pp. 98-101. Bangalore, India: IEEE, 2022. DOI: 10.1109/VLSID2022.2022.00030.
7. Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov. Software Architecture for Display Controller and Operating System Interaction. *RENSIT: Radioelektronika. Nanosistemy. Informacionnye Tehnologii*, 2021, 13(1):87-94. DOI: 10.17725/rensit.2021.13.087.
8. Bazhenov PS, Giatsintov AM, Mamrosenko KA. Approaches to providing data visualization on devices using modern real time operating systems. *Software & Systems,* 2021, 3:433-439. DOI: 10.15827/0236-235X.135.433-439.
9. Emmanuel Vadot. Adventure in DRMland Or How to Write a FreeBSD ARM64 DRM Driver. *Proceedings AsiaBSDCon*, 2019, pp. 9-13, (AsiaBSDCon. Tokyo, Japan: BSD Research, 2019). DOI: 10.25263/asiabsdcon2019/p01a.
10. Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov. Visualization of Graphic Information in General-Purpose Operating Systems. *RENSIT: Radioelektronika. Nanosistemy. Informacionnye Tehnologii*, 2019, 11(2):217-224e. DOI: 10.17725/rensit.2019.11.217.
11. Linux GPU Driver Developer's Guide. 2019. URL: https://dri.freedesktop.org/docs/drm/gpu/index.html (Access mode: 06.03.2019).
12. Verhaegen Luc. X and Modesetting: Atrophy Illustrated. 2006. URL: https://people.freedesktop.org/~libv/X_and_Modesetting_–_Atrophy_illustrated_(paper).pdf.
13. Kiyong Kwon, Dongwon Kang, Geon-Woo Ko, Seok-Young Kim, Seon-WookKim. Low-Cost Unified Pixel Converter from the MIPI DSI Packets intoArbitrary Pixel Sizes. *Electronics,* 2022, 11(8):1221. DOI:10.3390/electronics11081221.
14. Yeming Liu, Chengyue He. A Design of MIPI DSI Interface for LCD Display Driver. *Journal of Physics: Conference Series,* 2022, 2221(1):012015. DOI: 10.1088/1742-6596/2221/1/012015.
15. VESA and Industry Standards and Guidelines for Computer Display Monitor Timing (DMT), Version 1.0, Rev. 13. 39899 *Video Electronics Standards Association,* 2013, 105 p.

16. Hye-Hyun Lee, Yeon-Seob Song, Kang-Yoon Lee. Modeling of Nano-Scale PLL Using Verilog HDL. *Proc. 13th International Conference on Information and Communication Technology Convergence* (ICTC). IEEE, 2022, pp. 2101-2104. DOI: 10.1109/ICTC55196.2022.9952654.