DOI: 10.17725/rensit.2021.13.087

Архитектура программного обеспечения для задач взаимодействия контроллера вывода на экран и операционной системы

Пугин К.В., Мамросенко К.А., Гиацинтов А.М.

Центр визуализации и спутниковых информационных технологий. ОРВС ФГУ ФНЦ НИИ Системных исследований РАН, https://niisi.ru/

Москва 117218, Российская Федерация

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

Поступила 15.01.2021, рецензирована 22.01.2021, принята 29.01.2021

Представлена действительным членом РАЕН А.С. Дмитриевым

Аннотация: **В статье описана архитектура управляющего программного обеспечения для задач взаимодействия контроллера вывода на экран и операционной системы (ОС) Linux при наличии нескольких выводов на экран, частотами которых управляет один IP-блок синтезатора частоты (СЧ) с фазовой автоподстройкой. В ОС Linux не предусматривается специальных API для разработки драйверов для такого типа устройств, поэтому разработана новая производная модель представления контроллера вывода на экран внутри ядра Linux (Kernel Mode Setting, KMS) для корректной работы с такими устройствами. В статье описано несколько известных моделей для создания драйверов контроллеров вывода на экран, среди которых монолитная, компонентная и их комбинация. На базе компонентной модели создана новая производная модель, которая позволила решить проблемы “гонок” при взаимодействиях компонентных драйверов с единственным СЧ, при этом сохранив основные свойства базовой модели. Также описаны границы применимости разработанной модели при создании драйверов для низкоуровневых загрузчиков ОС (в т.ч. Linux). Произведена адаптация разработанной модели для загрузчиков. Модель обеспечивает взаимозаменяемость компонентов для работы с аппаратной частью при создании драйверов для Linux и загрузчика.**

Ключевые слова: **драйвер, встраиваемые системы, Linux, синтезатор частоты, загрузчик**

УДК 004.454

Благодарности: Работа выполнена в рамках государственного задания ФГУ ФНЦ НИИСИ РАН (Фундаментальные исследования 47 ГП) по теме № 0580-2021-0001 "Математическое обеспечение и инструментальные средства для моделирования, проектирования и разработки элементов сложных технических систем, программных комплексов и телекоммуникационных сетей в различных проблемно-ориентированных областях" (АААА-А19-119011790077-1).

Для цитирования: Пугин К.В., Мамросенко К.А., Гиацинтов А.М. Архитектура программного обеспечения для задач взаимодействия контроллера вывода на экран и операционной системы. *РЭНСИТ*, 2021, 13(1):87-94. 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 January 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: The work was carried out within the framework of the state assignment of FGU FSC NIISI RAS (Fundamental research 47 GP) on topic No. 0580-2021-0001 "Software and tools for modeling, design and development of elements of complex technical systems, software systems and telecommunication networks in various problem-oriented areas "(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.

Содержание

1. **Введение (88)**
2. **Материалы и методы. Модели разработки графических драйверов с использованием подсистемы DRM ОС Linux (89)**
3. **Результаты. Разработка новой модели драйвера для устройства с несколькими устройствами вывода и одним синтезатором частоты (91)**
4. **Обсуждение (92)**
5. **Заключение (93)**

Литература (93)

1.ВВЕДЕНИЕ

При разработке встраиваемых систем в ряде случаев необходимо принимать компромиссные решения для обеспечения соответствия изделия предъявляемым требованиям. Такая ситуация может возникнуть, к примеру, если имеются ограничения по тепловыделению или по размеру кристалла, либо потребуется уменьшение транзисторного бюджета кристалла.

Иногда таким компромиссным решением выступает уменьшение числа синтезаторов частоты (СЧ). Несколько контроллеров вывода на экран используют один и тот же СЧ, но при этом применяются различные протоколы взаимодействия с монитором (к примеру, TMDS и LVDS) при одновременной работе выводов.

Также в таких встраиваемых системах возможно использование общего конфигурируемого регистра. В таком регистре зачастую настраиваются как общие параметры всей подсистемы вывода на экран, так и некоторые параметры отдельных контроллеров. Это усложняет программирование подсистемы вывода, поскольку такой регистр становится общим ресурсом и требует управления доступом.

Такая схема системы вывода на экран требует разработки новых подходов к созданию драйверов для обеспечения ее работоспособности. Исходя из этого, перед нами стоят следующие задачи:

1. Рассмотреть возможные модели создания драйверов для таких встраиваемых систем.
2. Сравнить рассматриваемые модели по критериям расширяемости и работоспособности полученных драйверов.
3. Разработать модель драйвера, которая позволит решить задачу взаимодействия с операционной системой в случае нескольких контроллеров вывода на экран и одного синтезатора частоты.

Исследование будет проводиться на базе ОС Linux, так как это распространенная ОС для встраиваемых систем.

Существует несколько подсистем ядра,

применяемых при разработке драйвера контроллера вывода на экран под Linux. Применяемая модель обусловлена использованием той или иной подсистемы (она может быть приведена в документации либо определена на основе анализа драйверов-примеров). Рассмотрим наиболее популярные из них:

1. DRM. Как указанов [1], интерфейс контроллера вывода на экран и установки режимов является частью инфраструктуры Direct Rendering Manager (DRM). Разработчики DRM [2] рассматривают данную подсистему в виде 2 моделей — внешней и внутренней. Согласно их представлениям, во внешней модели есть 5 сущностей – framebuffer, plane, CRTC, encoder и connector, которые перечислены от близости к пользовательскому пространству до близости к выводу на экран. В модели внутренней подсистемы, кроме того, между encoder и connector существует bridge, а внутри connector может быть panel. В отличие от них, разработчики ПО [3] и независимые исследователи [4] рассматривают в основном только внешнюю модель либо упрощенную внешнюю модель.
2. FrameBuffer device (fbdev). В случае использования fbdev [4] модель состоит из 2 компонентов – взаимодействующего с пользовательским пространством буфера памяти и монолитного виртуального устройства DC (display controller), управляющего несколькими физическими. Настройка режимов этого устройства производилась напрямую из пользовательского пространства, что приводило к проблемам [5]. Наряду с обозначенными проблемами, такая упрощенная модель обладает низкой расширяемостью и меньшей функциональностью, чем модели на базе подсистемы DRM[6]. В текущий момент происходит переработка последних имеющихся драйверов на базе данной модели в драйверы на базе вариаций модели DRM[7].
3. Реализации драйверов под Android Display Framework (ADF) существуют, но обладают следующими недостатками: стандартные, независимые от аппаратной реализации, функции протокола необходимо разрабатывать с нуля в каждом драйвере, что увеличивает возможность появления ошибок и трудоемкость задачи. Также вследствие того, что подсистема ADF является развитием fbdev, возникает необходимость совмещения драйверов, поскольку, несмотря на атомарность, модели на базе этой подсистемы не ушли от монолитности, что и привело к ее замене на DRM [8].

2. МАТЕРИАЛЫ И МЕТОДЫ. МОДЕЛИ РАЗРАБОТКИ ГРАФИЧЕСКИХ ДРАЙВЕРОВ С ИСПОЛЬЗОВАНИЕМ ПОДСИСТЕМЫ DRM ОС LINUX

Реализации драйверов контроллеров в ряде других Unix-совместимых систем (к примеру, FreeBSD) также используют подсистему DRM, как наиболее расширяемую и функциональную на данный момент открытую подсистему для разработки данных драйверов [9].

Для разработки драйвера с использованием DRM необходимо использовать такой подход, чтоб модель драйвера не противоречила внешней и внутренней модели этой подсистемы, что позволит использовать API DRM для программирования контроллеров вывода на экран, особенно в части показа изображения на нескольких мониторах одновременно. В ходе разработки драйвера были проанализированы три модели.

Полностью монолитный тип (**рис. 1**)

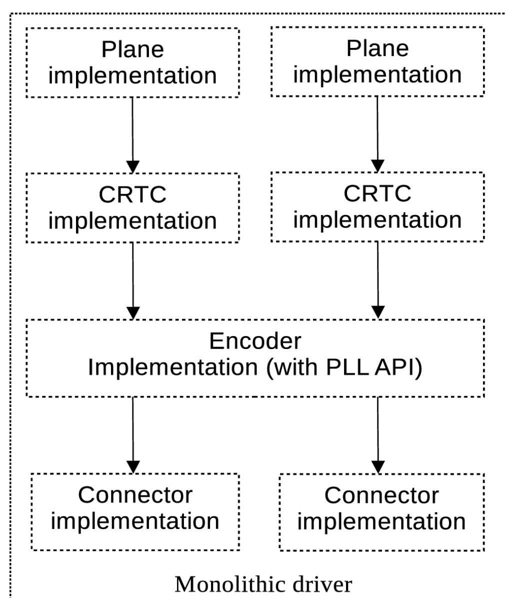

Рис. 1. *Монолитная модель драйвера.*

характеризуется относительной простотой устройства, но при этом имеет очень высокую связность. Из этого следует, что для различных подсистем DTLC (display transmitter link controller – контроллера, осуществляющего кодирование-декодирование сигнала для монитора и работу с протоколом передачи графических данных на монитор), а также для различных видов СЧ потребуется написание полностью различных драйверов, несмотря на идентичность IP-блока контроллера вывода на экран. Несмотря на формальную принадлежность такого драйвера подсистеме DRM, она наследует достоинства и недостатки подсистем предыдущего поколения. Это сильно ограничивает способность драйверов на базе этой модели к обновлению и замене компонентов – для каждой ревизии системы придется разрабатывать новый драйвер. В качестве плюса можно отметить простоту драйвера каждого устройства.

Второй тип – связная компонентная модель с учетом возможной замены аппаратной части (**Рис. 2**). Драйверы на базе такой модели имеют чуть меньшую связность компонентов, что упрощает доработку под новый тип контроллера, но не решает проблемы расширяемости и добавляет драйверу сложности из-за разнесения компонентов. Драйверы на базе этой модели, несмотря на внутреннее разделение на компоненты и более низкую связность, все равно не утрачивают важный недостаток полностью монолитных драйверов – требование модификации всего драйвера под каждую комбинацию компонентов всех возможных ревизий, что усложняет унификацию драйвера.

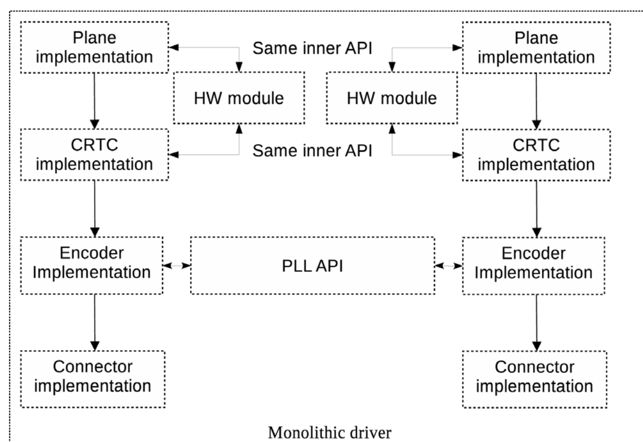

Рис. 2. *Частично монолитная модель драйвера.*

Третий тип построен на основе внешней модели DRM, а также на примере других драйверов для встраиваемых систем, использующих подсистему DRM [10] – модель, которая разработана по принципу максимальной независимости и взаимозаменяемости компонентов (**Рис. 3**). Драйверы на базе данной модели расширяемы и все их компоненты – взаимозаменяемые. Например, возможно заменить только драйвер DTLC, или только драйвер синтезатора частот, или только драйвер устройства конфигурации. Также возможно использование уже разработанных драйверов (с небольшими доработками для интеграции), что актуально при приобретении IP-блоков у других производителей. Также, вследствие отсутствия монолитности, возможно построение такого API, которое будет обращаться к общим устройствам, избегая «состояния гонки», и, вместе с тем, не интегрируя драйвер такого устройства в монолитную структуру.

Учет только внешней модели и максимальная независимость всех компонентов приводят к тому, что для управления некоторыми контроллерами DTLC требуется более сложный алгоритм, чем предусматривается абстракцией внешней модели. К примеру, требуется учет

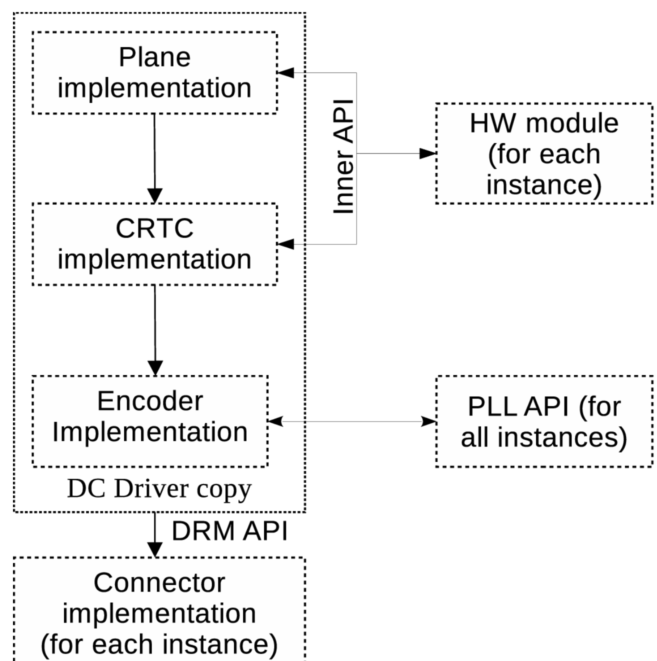

Рис. 3. *Компонентная модель драйвера.*

взаимодействия с компонентом DTLC при включении-выключении контроллера.

Поскольку многие современные протоколы взаимодействия с монитором, к примеру, DisplayPort, требуют использования указанного выше алгоритма, то необходима доработка представленной модели с учетом их особенностей.

3. РЕЗУЛЬТАТЫ. РАЗРАБОТКА НОВОЙ МОДЕЛИ ДРАЙВЕРА ДЛЯ УСТРОЙСТВА С НЕСКОЛЬКИМИ УСТРОЙСТВАМИ ВЫВОДА И ОДНИМ СИНТЕЗАТОРОМ ЧАСТОТЫ

Из представленных выше решений было опробовано компонентное (Рис. 3) как наиболее расширяемое и в большей степени задействующее внутренние механизмы ядра Linux. В ходе тестирования драйвера, построенного на решении 3, у данного решения были выявлены следующие недостатки:

1. «Состояние гонки» (race condition) при обращении к тактовому генератору. Поскольку с API управления тактовой частотой работают все копии, то в случае обращения к СЧ параллельно, частота будет меняться хаотичным образом, что приведет к миганиям изображения на экране и некорректной работе одного из выводов.
2. При использовании только внешней модели DRM, невозможно создать такой драйвер DTLC, который требует сложного взаимодействия с СЧ или с драйвером контроллера вывода на экран. Во внешней модели DRM заменяемым компонентом является connector, который не имеет динамических API для работы с DTLC. [2] Поэтому требуется использование внутренней модели DRM, несовместимой с моделью на рис. 3.
3. Экземпляры драйвера внутри ядра на базе компонентной модели полностью независимы и, вследствие этого, затруднена передача данных между ними. А поскольку копии идентичны, то при записи в конфигурационных регистр может образоваться состояние гонки.

Решение данных проблем потребовало

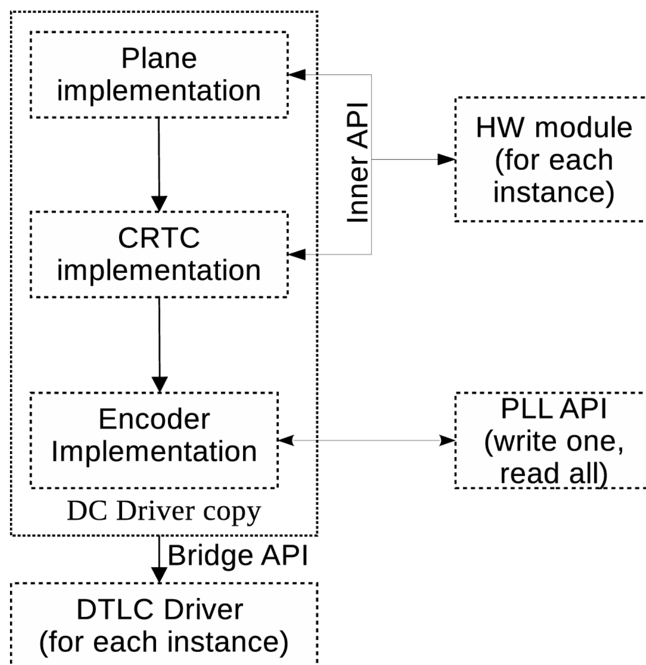

Рис. 4. *Разработанная модель драйвера.*

внесения изменений в компонентную модель, которая в итоге получила следующий вид (**Рис. 4**):

1. Первая из копий драйвера DC становится единственной копией, которая может сообщать информацию драйверу СЧ. Получать информацию могут все копии.
2. Первая же из копий драйвера определяет конфигурационные параметры, общие для всех копий, которые и устанавливает в общий конфигурационный регистр.
3. Для связи драйвера контроллера вывода на экран и драйвера DTLC используется компонент bridge внутренней модели DRM, который работает совместно с компонентом connector внешней модели.
4. Для расчета параметров допустимых режимов используется следующая формула:

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

где a_i – коэффициенты, связанные с DTLC (точное их значение зависит от аппаратной реализации DTLC), b – коэффициент погрешности (зависящий от протоколов взаимодействия, в частности, для DVI он составляет 0.025 [11]), а x_i – коэффициенты пиксельной частоты запрашиваемых режимов. При этом коэффициенты a_i и b – параметры, задающиеся при разработке драйвера на базе модели на рис. 4, а x_i – переменные, которые

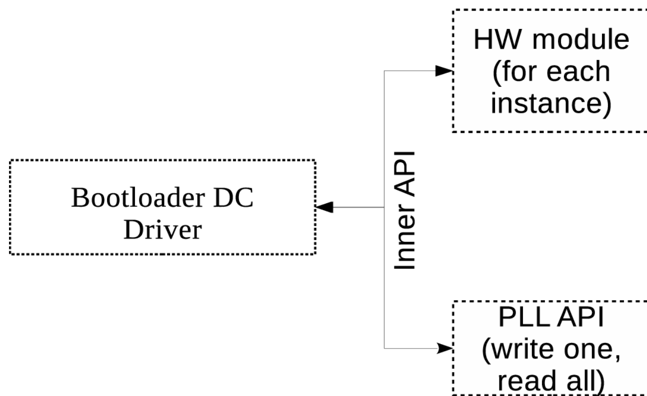

Рис. 5. *Модель драйвера низкоуровневого загрузчика.*

драйвер рассчитывает в процессе выполнения.

Большинство систем, работающих на Linux, используют низкоуровневые загрузчики (U-Boot[12], BareBox[13]) для первичной инициализации и загрузки устройств, в том числе и устройств подсистемы вывода на экран (**Рис. 5**). В таких случаях требуется разработка не только драйвера для ОС, но и драйвера загрузчика, который способен выполнять основные функции вывода изображения на экран. При некоторой модификации модели, обусловленной упрощенностью графического API загрузчика (при его наличии) и неполного соответствия этого графического API подсистеме DRM, возможно использование частей драйвера для ОС Linux на базе разработанной модели. Для использования в драйверах низкоуровневых загрузчиков (в частности, перечисленных выше) модель подверглась следующим модификациям:

1. Отдельный модуль DTLC в загрузчике не выделяется, так как не предусмотрена работа со сложными протоколами. Поэтому все вносится в драйвер контроллера вывода на экран.
2. Модель драйвера DC в загрузчике не предусматривает таких сущностей, как plane, encoder и crtc. Вся работа с выводом на экран происходит в одной и той же структуре, поэтому драйвер DC монолитен. Несмотря на сходство с моделью на базе подсистемы fbdev, подсистема загрузчика предусматривает меньше функций. Это обусловлено направленностью загрузчика на скорейшую загрузку ОС и уменьшение его размера.
3. Загрузчик не требует одновременного использования нескольких экранов, поэтому достаточно одной копии драйвера вывода на экран в случае одинаковых устройств вывода.

В графической подсистеме загрузчика очень редко встречается вариант одновременного использования СЧ несколькими устройствами вывода на экран, и остается только одновременное использование одного и того же участка памяти несколькими устройствами. Вследствие этого драйвера значительно упрощаются.

Интересным моментом также является взаимосвязь компонентов драйвера загрузчика и драйвера ОС. Если драйвер ОС построен на базе описанной в предыдущей главе модели, то его модуль взаимодействия с аппаратной частью будет применим и в загрузчике при условии совпадения внутреннего API. Это сильно упрощает написание драйверов сложных устройств, где взаимодействие с аппаратной частью требует алгоритмов расчета параметров (независимых от высокого уровня) или последовательностей записи в регистры устройства.

Несмотря на более простую структуру, драйвера загрузчика наследуют компонентный принцип. Это позволяет проводить инкрементное обновление компонентов драйвера как при изменении аппаратной части, так и при доработках загрузчика, упрощая его обновление [14].

4. ОБСУЖДЕНИЕ

Разработанная модель имеет следующие преимущества перед рассмотренными вариантами моделей:

1. Расширяемость – драйвер DC на базе этой модели может быть расширен любым драйвером DTLC (в том числе и разработанным до создания драйвера на базе представленной выше модели), который использует bridge API.
2. Ускорение перехода на новые модели контроллеров. Поскольку для перехода на новую модель контроллера требуется только разработка нового модуля взаимодействия с аппаратной частью, то, в отличие от модели на рис. 1, не требуется полная

переработка драйвера, а в отличие от модели 2 – не требуется полное обновление драйвера (достаточно только обновить модуль взаимодействия с аппаратной частью).

3. В отличие от драйверов на базе компонентной модели, представленной на рис. 3, драйверы на базе разработанной модели не входят в состояние гонки, если прикладное ПО постоянно обращается к DRM API для смены частоты.
4. Также в отличие от модели на рис. 3, драйвера на базе доработанной модели способны показывать несколько изображений на всех мониторах одновременно (при установке допустимых режимов). Драйвер, построенный по модели 3, будет давать нестабильное изображение в случае постоянной записи в СЧ (поскольку разные мониторы требуют разные пиксельные частоты, изображение будет мигать), а драйверы по моделям 1 и 2 выставят частоту того монитора, что был подключен последним).

В отличие от рассмотренных моделей, разработанная модель имеет следующие недостатки:

1. Повышенную сложность разработки драйверов по сравнению с моделью 1.
2. Больше возможностей для отказа вследствие большего числа точек отказа (возможности большего числа копий драйвера и их произвольной рекомбинации с драйверами DTLC).
3. Использование Bridge API затруднено на архитектурах, где отсутствует device tree, там необходимо производить сложные манипуляции с данными для поиска bridge.

5. ЗАКЛЮЧЕНИЕ

В ходе данной работы были проанализированы возможные варианты архитектуры драйвера для устройства с несколькими контроллерами вывода на экран, но одним синтезатором частот. На основе нескольких известных моделей драйверов была разработана новая производная модель драйвера для такого устройства, которая совмещает достоинства всех предложенных моделей, но имеет один недостаток — создание драйверов на ее базе затруднительно без использования device-tree. Данные недостаток малосущественен для встраиваемых систем с Linux, поскольку большинство из них используют device tree.

Апробация созданной модели производилась при разработке драйвера для устройства с выводами DVI и LVDS на одном СЧ, на базе Linux, архитектуры MIPS. Было произведено тестирование реализаций всех моделей с использованием системы прототипирования Protium [15], прошедшее успешно.

Результаты данной работы могут быть применены при создании новых драйверов для такого класса устройств для Unix-подобных систем в случае использования аппаратной части с несколькими выводами и одним синтезатором частоты.

ЛИТЕРАТУРА

1. Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov. Visualization of graphic information in general-purpose operating systems. *RENSIT*, 2019, 11(12):217-224. DOI: 10.17725/rensit. 2019.12.217.
2. *Linux GPU Driver Developer's Guide* [электронный ресурс]. URL: https://dri.freedesktop.org/docs/drm/gpu/index.html (дата обращения: 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 [электронный ресурс]. URL: https://www.phoronix.com/scan.php?page=news_item&px=FBDEVDRM-DRM-Over-FBDEV

(дата обращения: 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 [электронный ресурс]. 1999. URL: https://web.archive.org/web/20120813201146/http://www.ddwg.org/lib/dvi_10.pdf (дата обращения: 30.12.2020).
12. Anatolij Gustschin. U-Boot video API [электронный ресурс]. URL: git https://gitlab.denx.de/u-boot/custodians/u-boot-video.git (дата обращения: 29.07.2020).
13. Hauer S. BareBox [электронный ресурс]. URL: https://github.com/saschahauer/barebox (дата обращения: 29.07.2020).
14. Kang Y, Chen J, Li B. Generic Bootloader Architecture Based on Automatic Update Mechanism. I*EEE 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.

Пугин Константин Витальевич
программист
НИИ Системных исследований РАН
36/1, Нахимовский просп., Москва 117218, Россия
rilian@niisi.ras.ru

Мамросенко Кирилл Анатольевич
к.т.н.
НИИ Системных исследований РАН
36/1, Нахимовский просп., Москва 117218, Россия
mamrosenko_k@niisi.ras.ru

Гиацинтов Александр Михайлович
к.т.н.
НИИ Системных исследований РАН
36/1, Нахимовский просп., Москва 117218, Россия
algts@niisi.ras.ru.