УДК 004.454
DOI: 10.15827/0236-235X.123.425-429
Дата подачи статьи: 28.04.18
2018. Т. 31. № 3. С. 425–429

Методы разработки драйверов графической подсистемы

И.А. Ефремов [1], *программист, efremov@niisi.ras.ru*
К.А. Мамросенко [1], *к.т.н., руководитель Центра, mamrosenko_k@niisi.ras.ru*
В.Н. Решетников [1], *д.ф.-м.н., профессор, главный научный сотрудник, rvn_@mail.ru*

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

В статье описаны проблемы разработки ПО для задач взаимодействия систем на кристалле и ОС Linux. Архитектурой ОС предусмотрено создание драйвера – компонента, обеспечивающего прием и передачу данных устройства с использованием программного интерфейса. Разработка драйверов для ОС с открытым исходным кодом затрудняется из-за непрерывных изменений функций и структуры ядра.

Описаны структура и составные части графической подсистемы, представляющей собой набор компонентов, находящихся в разных адресных пространствах доступа виртуальной памяти ОС и взаимодействующих между собой, в том числе посредством интерфейса системных вызовов. Программирование графического ядра выполняется при помощи заполнения буфера команд: для каждого приложения создается контекст графического ядра, содержащий свой командный буфер и все необходимые данные, используемые графическим ядром для отрисовки/расчетов, – координаты, векторы нормали, цвета, текстуры.

Существуют несколько подходов к установке графического режима, однако наиболее оправданным решением является применение модуля KMS (Kernel Mode Setting), который используется ключевыми производителями микропроцессоров и графических карт. Для полной реализации возможностей графического ядра необходимо обеспечить взаимодействие модулей ядра ОС и пространства пользователя посредством создания собственных системных вызовов, регламентирующих низкоуровневую работу с устройством.

Применение платформ прототипирования на основе FPGA-матриц позволяет проверить работоспособность ПО, получить некоторые характеристики производительности и выявить ошибки в системе на кристалле на ранних стадиях проектирования. Отладка модулей ядра занимает значительное время в силу ограничений, накладываемых со стороны как платформы для прототипирования, так и ОС. Кроме того, ошибки, возникающие в коде ядра, трудновоспроизводимы, что также затрудняет отладку модулей ядра.

В статье рассмотрены подходы к реализации KMS-модуля и компонентов графической подсистемы ОС Linux, которые позволяют обеспечить корректное взаимодействие ОС и контроллера вывода на экран системы на кристалле.

Ключевые слова: *графическое ядро, драйвер, Linux, СнК, разработка, модуль ядра.*

В настоящее время одной из задач микроэлектроники является создание *систем на кристалле* (СнК) с существенными ограничениями по тепловыделению и энергопотреблению [1]. Проектирование СнК включает в себя разработку ПО для решения задач взаимодействия оборудования и ОС [2]. Работа с трехмерной графикой в реальном масштабе времени, как правило, требует наличия графического ядра в СнК.

Архитектурой ОС предусмотрено создание компонента – драйвера, обеспечивающего прием и передачу данных устройства с использованием программного интерфейса [3]. При разработке драйверов для ОС с открытым исходным кодом, например, для ядра ОС Linux, возможно возникновение определенных трудностей из-за непрерывных изменений функций и структуры ядра [4]. Кроме того, количество источников информации по данному вопросу, находящихся в открытом доступе, невелико.

В настоящей статье приведены методы разработки драйверов графической подсистемы, обеспечивающих и установку графического режима отображения, рассмотрены проблемы, связанные с отладкой модулей ядра.

Графическая подсистема ОС Linux

Обработку 3D-графики обеспечивает графическая подсистема (или графический стек) ОС Linux [5]. Графический стек представляет собой набор компонентов, находящихся в разных адресных пространствах доступа виртуальной памяти ОС и взаимодействующих между собой посредством интерфейса системных вызовов [6] (рис. 1).

В пользовательском пространстве на верхнем уровне набор функций и их поведение описаны спецификацией OpenGL (Open Graphics Library). Свободную реализацию графического API (Application Programming Interface) OpenGL в ОС Linux представляет библиотека Mesa 3D, включающая в себя набор драйверов 3D-графики в пользовательском пространстве. Библиотека Mesa реализует независимый от поставщиков драйверов кросплатформенный стандартный API-интерфейс для взаимодействия с графическими ускорителями различных производителей.

Библиотека libdrm реализует интерфейс DRM в пространстве пользователя, применяя системные вызовы ioctl [7]. Для обеспечения взаимодействия модуля высокого уровня OpenGL и библиотеки

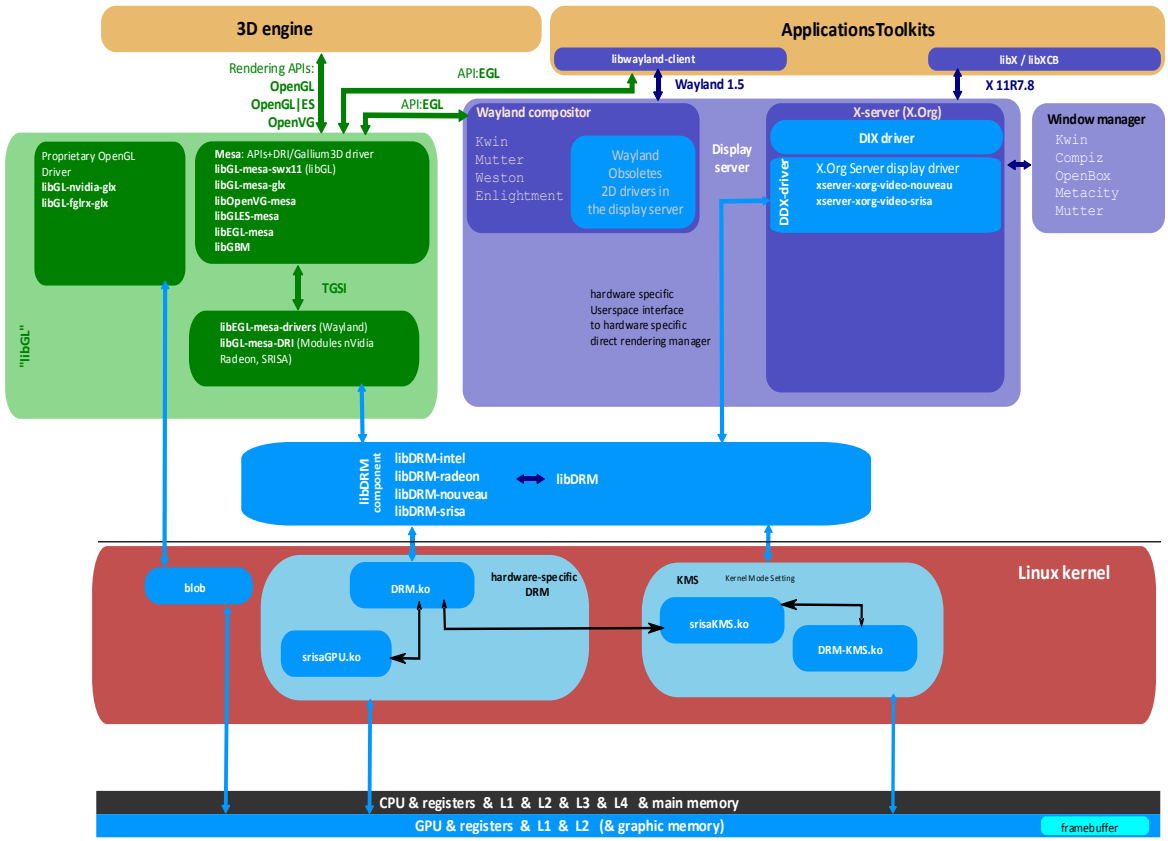

Рис. 1. Компоненты графического стека ОС Linux

Fig. 1. Components of OS Linux graphics stack

libdrm может быть использована библиотека создания драйверов Gallium3D.

Gallium3D является частью Mesa и направлен на упрощение разработки драйверов 3D-графики. Наличие Gallium3D API подразумевает использование TGSI (Tungsten Graphics Shader Infrastructure), являющегося промежуточным представлением описания шейдеров.

После предварительной обработки входящих данных в библиотеке OpenGL, включающих информацию об объектах прорисовки и шейдерных подпрограммах, производится преобразование данных в промежуточный байт-код. На этом этапе также осуществляется оптимизация операций, выполняемых в шейдерных программах, например, разворачивание циклов обработки. Далее сгенерированный байт-код передается в компилятор, преобразующий его в машинные инструкции, предназначенные для конкретного графического процессора, а также формирующий буфер команд. Программирование графического ядра выполняется при помощи заполнения буфера команд: для каждого приложения создается контекст графического ядра, содержащий свой командный буфер и все необходимые данные, используемые графическим ядром для отрисовки/расчетов, – координаты, векторы нормали, цвета, текстуры. Графическое ядро не располагает собственной памятью, поэтому буферы формируются в системной памяти.

Драйвер должен поддерживать многопоточное использование, где на каждый поток выделяются один буфер контекста и один буфер команд. За счет наличия только одного буфера команд синхронизация требуется лишь на момент передачи буфера в графическое ядро. На момент заполнения буфера она не требуется. Правильное выставление контекста в ядре для каждого потока должно гарантироваться драйвером графической подсистемы. Также определяются доступные операции для выполнения на конкретной модели графического процессора, ограничения для входящих данных, например, поддерживаемые форматы сжатия текстур.

В пространстве ядра модуль Direct Rendering Manager (DRM) предоставляет API, который используют программы пользовательского пространства для отправки команд и данных графическому ядру и настройки некоторых параметров режима отображения. DRM предоставляет только базовую функциональность, с которой могут работать различные драйверы, а также снабжает пользовательское пространство неким минимальным набором системных вызовов input output control (ioctl) с общей, независимой от оборудования функциональностью.

Существуют несколько подходов к установке графического режима. Установка графических режимов из пользовательского пространства (User Mode-Setting) имеет проблемы:

– неоднократная инициализация оборудования (BIOS, framebuffer и X-сервер);

– некорректное отображение при переключениях между виртуальными консолями и экземплярами X-сервера и в процессе загрузки;

– дублирование кода драйвера (fbdev и драйверов графики для X-сервера).

Решением данных проблем является использование модуля KMS (Kernel Mode Setting) [8]. Драйвер контроллера вывода на экран DC (Display Controller) находится в пространстве ядра Linux и также должен включать функционал KMS-модуля.

Kernel Mode Setting

Ioctl обычно используются для выполнения над устройством некоторых специфических (управляющих) действий, которые не обеспечиваются регулярными POSIX-вызовами (read(), write(), lseek() и др.) [9]. Часто это могут быть действия, зависящие от конкретных аппаратных особенностей реализации устройства. Для использования возможностей графического ядра в полном объеме необходимо обеспечить взаимодействие модулей ядра ОС и пространства пользователя посредством создания собственных системных вызовов, регламентирующих низкоуровневую работу с устройством (ioctl). На данный момент наиболее актуально применение KMS API для установки графического режима на контроллере вывода на экран. Данное API используется ключевыми производителями микропроцессоров и графических карт.

При построении KMS-модуля необходимы инициализация структур из KMS API, а также установление связей между блоками модуля. DRM-коннектор представляет собой совокупность структур и функций, которые необходимо реализовать в собственном драйвере. Коннектор обеспечивает получение данных от подключенного дисплея и проверку корректности задаваемых графических режимов.

Кадровый буфер (Frame buffer) служит источником данных для дисплей-контроллера. Он представляет собой объекты памяти (memory objects), которые передаются конвейеру отображения CRTC для вывода на экран (рис. 2).

Плоскость (plane) содержит информацию о параметрах изображения и ссылку на дополнительный кадровый буфер. Плоскость передается в CRTC для объединения изображения основного кадрового буфера с изображением дополнительного буфера. CRTC представляет собой общий конвейер отображения, принимая данные пикселей из плоскостей и смешивая их вместе, а также регламентируя временные интервалы отображения (display timing).

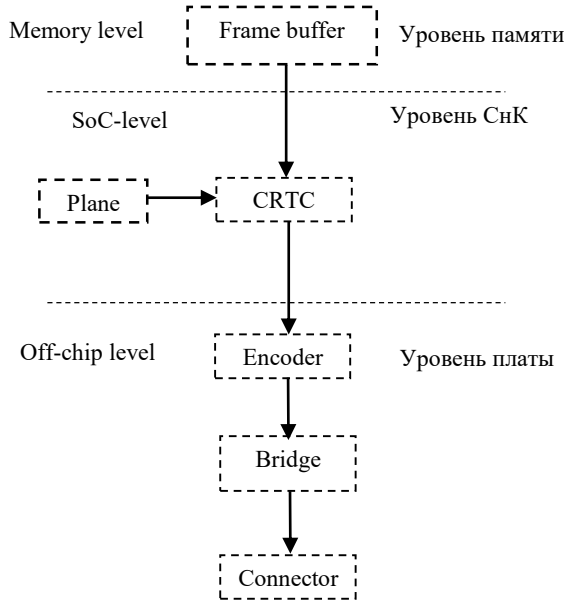

Рис. 2. Схема KMS-модуля

Fig. 2. KMS module scheme

Графический контроллер не располагает собственной памятью, поэтому буферы формируются в системной памяти. Менеджер графической памяти (GEM) управляет выделением памяти, чтением и записью в буферы. Системные вызовы, являющиеся частью драйвера, обеспечивают создание объектов памяти. Более того, фактически создание буфера для драйверов, использующих GEM, выполняется с помощью специфичного для драйвера ioctl – GEM имеет только общий интерфейс для совместного использования и уничтожения объектов. Для задач, не требующих всей гибкости работы с памятью, например, выделение кадровых буферов для KMS, применяется API для работы с dumb-буферами. Буферы связываются в список, который автоматически опрашивается контроллером на наличие новых буферов.

Энкодер (encoder) является преобразователем изображения в требуемый выходной формат, например, цифровой или аналоговый. На сегодняшний день использование энкодеров считается устаревшим подходом, вместо них рекомендуется применять компоненты bridge. Основное отличие bridge от энкодера – недоступность из пространства пользователя.

В настоящее время рекомендуется применять атомарный подход к установке графического режима. В отличие от применяемого ранее транзакционного подхода атомарный подход обеспечивает возможность проверки корректности режимов работы без изменения состояния аппаратных средств. При атомарном подходе общее состояние драйвера определяется состоянием трех компонентов

KMS-модуля с использованием структур состояний: *drm_plane_state*, *drm_crtc_state* и *drm_connector_state*. Структура *drm_atomic_state* – глобальное состояние объекта для атомарных обновлений, включает в себя эти структуры состояний.

Файл описания аппаратной части

В ОС Linux предусмотрена возможность создания .dts-файла (Device Tree Source). Преимущество использования dts-файла в том, что разработчику модулей ядра не требуется перекомпилировать ядро для различных платформ, содержащих СнК. Файл описания используемой архитектуры и блоков СнК dtsi представлен в виде дерева узлов и служит в основном для предоставления информации ядру ОС. Разработчик, предоставляя дерево устройств, делает возможным более широкую поддержку различных конфигураций оборудования в рамках архитектурной линейки СнК.

Файлы дерева устройств могут быть разделены на несколько частей в нескольких файлах. Файлы уровня платы dts включают в себя dtsi-файлы, описывающие уровень СнК. При создании нового дерева устройств для аппаратной платформы необходимо полностью описывать свойства устройства. Файл дерева устройств хранит и данные о свойствах подключения коннектора видеоинтерфейса.

Отладка модулей ядра

В настоящее время разработка и отладка драйверов зачастую происходят параллельно с разработкой СнК. Как правило, в этих целях используют платформы прототипирования на основе FPGA-матриц. Платформы прототипирования позволяют проверить работоспособность ПО, получить некоторые характеристики производительности и выявить ошибки в СнК на ранних стадиях производства [10].

Однако даже при использовании современных программных и аппаратных средств отладка модулей ядра занимает значительное время из-за ограничений, накладываемых и платформой для прототипирования (тактовая частота, длительное время сборки проекта прошивки), и модулями ядра (многие ядерные механизмы принципиально существуют только во временных зависимостях и не могут быть приостановлены). Кроме того, ошибки, возникающие в коде ядра, трудновоспроизводимы, что также затрудняет отладку модулей ядра.

Запуск тестовых приложений с трехмерной графикой происходит с использованием моделирующего комплекса, зачастую не имеющего модуля вывода на экран. В таком случае для проверки корректности визуализации используется сохранение содержимого кадрового буфера в файлы. Для этого на этапе инициализации тестовой программы используется флаг EGL_PBUFFER_BIT – он дает возможность установить конфигурацию системы, не требующей использования физического дисплея для вывода изображения и позволяющей прорисовывать изображения в системную память.

Заключение

При проектировании СнК необходим комплексный подход, учитывающий особенности платформ прототипирования и архитектуру графической подсистемы ОС. Поддержка разрабатываемого ПО в течение всего жизненного цикла СнК является необходимым процессом в условиях изменения архитектуры ядра ОС с открытым исходным кодом. Поддержка трехмерной графики требует от разработчика ПО построения согласованной графической подсистемы с набором компонентов, находящихся в различных адресных пространствах доступа виртуальной памяти ОС. Создание модулей графической подсистемы ОС Linux – актуальная и трудоемкая задача, которой в настоящее время посвящается все больше работ в открытой печати. Однако требуются новые инструменты разработки и отладки. В настоящее время отладка модулей ядра занимает значительное время из-за ограничений со стороны ОС и средств прототипирования.

Рассмотренные подходы к реализации KMS-модуля и компонентов графической подсистемы ОС Linux позволяют обеспечить корректное взаимодействие ОС и контроллера вывода на экран СнК. Планируются внедрение новой функциональности KMS-модуля, а также разработка драйверов пользовательского пространства.

Литература

1. Бобков С.Г. Высокопроизводительные вычислительные системы. М.: Изд-во НИИСИ РАН, 2014. 296 с.

2. Aryashev S.I., Rogatkin B.Y., Barskikh M.E. Modern methods of functional verification of rtl units of VLSI microprocessor // Проблемы разработки перспективных микро- и наноэлектронных систем МЭС. 2015. № 2. С. 119–122 (англ.).

3. Решетников В.Н. Космические телекоммуникации. Системы спутниковой связи и навигации. СПб: Ленинградское изд-во, 2010. 132 с.

4. Гиацинтов А.М., Баженов П.С. Визуализация виртуальных трехмерных сцен на однокристальных системах // Программные продукты, системы и алгоритмы. 2017. № 3. URL: http://swsys-web.ru/virtual-three-dimensional-scenes-on-single-chip-systems.html (дата обращения: 20.04.18).

5. Лав Р. Разработка ядра Linux. СПб: Вильямс, 2006. 448 с.

6. Venkateswaran S. Essential Linux Device Drivers. Prentice Hall, 2008, 714 с.

7. The Linux kernel documentation. URL: https://www.kernel.org/doc/html/v4.11/gpu/drm-kms.html (дата обращения: 20.04.2018).

8. Madieu J. Linux device drivers development. Packt Publishing Ltd, 2017, 586 с.

9. Bovet D.P., Cesati M. Understanding the Linux Kernel. USA, O'Reilly Media, 2005, 702 p.

10. Богданов А.Ю. Опыт применения платформы прототипирования на ПЛИС «Protium» для верификации микропроцессоров // Тр. НИИСИ РАН. 2017. Т. 7. № 2. С. 46–49.

Software & Systems
DOI: 10.15827/0236-235X.123.425-429

Received 28.04.18
2018, vol. 31, no. 3, pp. 425–429

Methods of developing graphics subsystem drivers

I.A. Efremov [1], *Programmer, efremov@niisi.ras.ru*
K.A. Mamrosenko [1], *Ph.D. (Engineering), Head of the Center, mamrosenko_k@niisi.ras.ru*
V.N. Reshetnikov [1], *Dr.Sc. (Physics and Mathematics), Professor, Chief Researcher, rvn_@mail.ru*

[1] *Center of Visualization and Satellite Information Technologies SRISA, Moscow, 117218, Russian Federation*

Abstract. The paper describes problems of software development for the problems of interaction between systems-on-a-chip (SoC) and the Linux operating system (OS). The OS architecture provides various instruments for creating a driver that is a component allowing the device data exchange using a software interface. The development of drivers for an open source OS is difficult due to continuous changes in functions and a kernel structure.

The paper describes graphics subsystem structure and components. The subsystem is a component kit located in different address spaces of OS virtual storage. The components interact through a system call interface. Programming of a graphics engine is performed by filling a command buffer. Each application has a graphics engine context that contains its own command buffer and all necessary data used by the graphics engine for rendering/calculations: coordinates, normal vectors, colors, textures.

There are several approaches to setting graphics mode. However, the most reasonable solution is using KMS module (Kernel Mode Setting). Key manufacturers of microprocessors and graphics cards commonly use these modules. It is necessary to ensure the interaction between OS kernel modules and user space through creating own specific system calls. These system calls regulate low-level operations with the device and allow taking full advantage of the graphics unit capabilities.

Using FPGA-based prototyping platforms allows verifying software functionality, getting performance characteristics and finding errors in SoC hardware design at early stages. Debugging kernel modules is time-consuming due to limitations imposed both by a prototyping platform and the OS. In addition, the errors in a kernel code are difficult to reproduce, which also complicateы debugging of kernel modules.

The paper considers some approaches to implementation of Linux OS KMS module and graphics subsystem components, which provide correct interaction of the OS and the SoC display controller.

Keywords: GPU, driver, Linux, SoC, development, kernel module.

References

1. Bobkov S.G. *High Performance Computing Systems*. Moscow, 2014, SRISA RAS Publ., 296 p.
2. Aryashev S.I., Rogatkin B.Y., Barskikh M.E. Modern methods of functional verification of rtl units of VLSI microprocessor. *Development Problems of Advanced MES Micro- and Nanoelectronic Systems*. 2015, no. 2, pp. 119–122 (in Russ.).
3. Reshetnikov V.N. *Space Telecommunications. Satellite Communication and Navigation Systems*. St. Petersburg, Leningradskoe izdatelstvo Publ., 2010, 132 p.
4. Giatsintov A.M., Bazhenov P.S. Visualization of virtual three-dimensional scenes on single-chip systems. *Software, Systems and Algorithms*. 2017, no. 3. Available at: http://swsys-web.ru/virtual-three-dimensional-scenes-on-single-chip-systems.html (accessed April 20, 2018).
5. Love R. *Linux Kernel Development*. Sams Publ., 2003, 332 p.
6. Venkateswaran S. *Essential Linux Device Drivers*. Prentice Hall Publ., 2008, 714 p.
7. *The Linux Kernel Documentation*. Available at: https://www.kernel.org/doc/html/v4.11/gpu/drm-kms.html (accessed April 20, 2018).
8. Madieu J. *Linux Device Drivers Development*. Packt Publ. Ltd, 2017, 586 p.
9. Bovet D.P., Cesati M. *Understanding the Linux Kernel*. 3rd ed., O'Reilly Publ., 2005, 702 p.
10. Bogdanov A.Yu. Using a prototype platform on the FPGA "Protium" for microprocessor verification. *SRISA RAS Proc*. 2017, vol. 7, no. 2, pp. 46–49 (in Russ.).

Примеры библиографического описания статьи

1. Ефремов И.А., Мамросенко К.А., Решетников В.Н. Методы разработки драйверов графической подсистемы // Программные продукты и системы. 2018. Т. 31. № 3. С. 425–429. DOI: 10.15827/0236-235X.123.425-429.

2. Efremov I.A., Mamrosenko K.A., Reshetnikov V.N. Methods of developing graphics subsystem drivers. *Software & Systems*. 2018, vol. 31, no. 3, pp. 425–429 (in Russ.). DOI: 10.15827/0236-235X.123.425-429.