УДК 004.921
DOI: 10.15827/2311-6749.18.4.1

ИССЛЕДОВАНИЕ И ОТЛАДКА КОМПОНЕНТОВ ДЛЯ ОБРАБОТКИ ТРЕХМЕРНОЙ ГРАФИКИ ОПЕРАЦИОННЫХ СИСТЕМ

Баженов П.С., ведущий программист, bps@niisi.ras.ru;
Мамросенко К.А., к.т.н., руководитель Центра, mamrosenko_k@niisi.ras.ru;
Решетников В.Н., д.ф.-м.н., профессор, главный научный сотрудник, rvn_@mail.ru
(Центр визуализации и спутниковых информационных технологий, ФНЦ НИИ системных исследований РАН, Нахимовский просп., 36, к. 1., г. Москва, 117218, Россия)

В статье дано общее описание графического стека в ОС Linux, а также отдельных его компонентов. Рассмотрены процессы взаимодействия между компонентами, расположенными в ядре ОС и в пользовательском пространстве. Описываются инструментарий, с помощью которого производятся сборка и отладка драйверов, а также этапы настройки окружения и сборки драйверов с различными конфигурациями. Приводятся подход к тестированию драйверов и способы получения результатов с использованием системы прототипирования Protium с учетом особенностей данной системы.

Ключевые слова: *разработка, драйвер, Linux, графический стек, MIPS, OpenCL, Protium, OpenGL.*

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

Для взаимодействия между ОС Linux и графическими устройствами разрабатывается ПО – драйверы. Ядро Linux входит в множество современных ОС, используемых большим количеством пользователей, для которых важны их надежность и стабильность. Также появляются новые подходы и стандарты для обеспечения визуализации с помощью графического оборудования. Поэтому так актуальна задача разработки графических драйверов под ОС Linux.

Графический стек

Графический стек в ОС Linux состоит из компонентов, которые располагаются как в защищенной памяти системы – в пространстве ядра (kernel-space), так и в незащищенной области – в пространстве задач (user-space). Компоненты в пространстве ядра используются для взаимодействия с регистрами CPU, регистрами GPU, кэш-памятью различных уровней, оперативной и видеопамятью и т.д. Выделяемая память в этом пространстве используется ядром, расширениями ядра и модулями для взаимодействия с устройствами. Как правило, для работы с графическими устройствами используются компоненты ОС Linux DRM и KMS. Ряд производителей аппаратного обеспечения для работы с графикой используют проприетарные модули ядра (blob). DRM (Direct Rendering Manager) – подсистема ядра Linux, предоставляющая API, с помощью которого программы из пространства задач могут использоваться для отправки команд и данных на графический процессор. Blob – это бинарный модуль с закрытым исходным кодом для использования в ОС с открытым кодом.

ОС Linux предоставляет несколько подходов управления выводом информации на экран. Одним из них является использование fbdev (framebuffer device). Он представляет кадровый буфер какого-либо графического устройства и позволяет приложению получить доступ к этому устройству через интерфейс, поэтому приложение не нуждается в прямом обращении к низкоуровневой части [2]. В настоящее время считается устаревшим и не рекомендуется к применению. Более современным подходом является применение KMS (kernel mode-setting). KMS позволяет задать параметры (разрешение экрана, глубина цвета, частота обновления) для контроллера вывода на экран на уровне ядра.

Пространство задач используется для обеспечения работы пользовательского ПО и пользовательских компонентов драйверов. Компоненты графического драйвера в этом пространстве представляются файлами библиотек, в которых реализуется программный интерфейс для других приложений. К таким компонентам можно отнести библиотеку libdrm, реализацию графического API (libGL, libGLES), реализацию интерфейса EGL. Библиотека libdrm реализует интерфейс DRM в пространстве пользователя, используя системные вызовы ioctl [1]. EGL – это интерфейс между API OpenGL ES или OpenVG и базовой платформой оконной системы (X11, wayland). Он обеспечивает управление графическим контекстом, привязкой поверхностей/буферов, синхронизацией рендеринга и позволяет осуществлять высокопроизводительный 2D- и 3D-рендеринг с использованием других API Khronos. Также EGL предоставляет возмож-

ность передачи данных между API, например, между видеоподсистемой, использующей OpenMAX AL, и графическим процессором, работающим с OpenGL ES [3].

Также в пространстве задач находится такой компонент, как X11 (X Window System) – это оконная система для построения графического пользовательского интерфейса (GUI). Модель «клиент–сервер» в X-системе отлична от типовой модели, где клиент работает на локальном компьютере и запрашивает службы с сервера. В X-системе сервер запускается на локальном компьютере и предоставляет возможность использования дисплея и сервисов для клиентских программ. Клиентские программы могут быть запущены локально или удаленно с использованием сети.

Существует как проприетарная (закрытая), так и свободная реализация графических драйверов. К примеру, к проприетарным относятся драйверы от AMD или NVIDIA, а к свободным – MESA.

Компилятор и кросс-компиляция

Для сборки драйверов Linux используется инструментарий (toolchain), состоящий из нескольких компонентов [4]. Первым компонентом являются бинарные утилиты (binutils), в него входят утилиты (бинарные программы, библиотеки), необходимые для сборки и компиляции программ. Двумя основными утилитами являются *as* (ассемблер) и *ld* (линкер). Утилита *as* переводит код ассемблера, генерируемый GCC, в бинарный файл. Утилита *ld* связывает объектные коды в библиотеки или в исполняемые файлы.

Следующим компонентом, входящим в toolchain, является GCC (GNU Compiler Collection) – набор компиляторов для различных языков программирования (C, C++, Java и т.д.). Изначально GCC содержал только компилятор языка C и аббревиатура GCC расшифровывалась как GNU C Compiler. Со временем набор компиляторов расширялся под другие языки программирования. GCC является ключевым компонентом в GNU toolchain, и в сборке драйверов используется C-компилятор.

В состав toolchain также входит стандартная библиотека языка C (GNU C library), содержащая набор заголовочных файлов с описанием функций и библиотек, соответствующих стандарту ANCI C.

Компонентами toolchain являются также утилита make – инструмент для управления генерацией исполняемых файлов программ и библиотек из файлов исходных кодов, и GDB – утилита для отладки программ и анализа действий выполнения программы.

Бывают случаи, когда необходимо собрать программу под архитектуру, отличную от архитектуры сборочной машины, например, ARM или MIPS. Для этих целей применяется кросс-компиляция. Принцип кросс-компиляции заключается в том, что существуют две различные системы: система, на которой запускают инструменты компиляции (host система), и система, для которой эти инструменты генерируют код (target-система). Компилятор, собирающий программы для другой архитектуры, называется компилятором кросс-компиляции, или кросс-компилятором.

Подготовка к сборке

На сборочной машине должно находиться предварительно скомпилированное ядро Linux, которое будет запускаться на отладочном стенде (системе). В модуле графического ядра рекомендуется указать размер непрерывной физической графической памяти. Для эффективной работы графического драйвера данная область должна быть достаточно большой. Непрерывная графическая память должна вмещать два буфера: кадровый и глубины. Объем необходимой непрерывной памяти также зависит от используемого графического приложения. Например, для отображения в режиме HD 1080 потребуется около 16 Мбайт в зависимости от конфигурации.

Далее идут подготовка и настройка сборочного окружения (build environment). Для удобства выделяется директория под проект с драйверами. В данную директорию распаковываются исходный код графических драйверов, которые необходимо собрать. После распаковки нужно определить ряд параметров: какой используется компилятор (или кросс-компилятор), место расположения исходного кода нужного ядра, место установки скомпилированных бинарных файлов, место расположения инструментария (toolchain), а также (опционально) пути к сторонним библиотекам, требуемые для сборки модулей драйвера, например, для использования функционала X11. Эти параметры задаются с помощью команды export перед сборкой. Для удобства можно создать файл-сценарий, в котором будут задаваться параметры окружения.

Сборка драйверов

Для выполнения сборки драйвера используется файл-сценарий, который содержит в себе две секции: конфигурации под архитектуры и параметры драйвера. В первой секции задаются конфигурации сборки под определенную архитектуру микропроцессора с вышеописанными параметрами и условиями. Этих конфигураций может быть несколько, если требуется выполнить сборку для нескольких архитектур.

Выбор архитектуры осуществляется при запуске файла-сценария флагом, указанным в конфигурации. Во второй секции находятся все возможные параметры сборки драйверов, используемые в make-файлах. Например, к ним могут относиться следующие: сборка драйвера с отладочной информацией, поддержка 2D/3D, поддержка DRI/DRI3, поддержка FB, поддержка OpenCL и т.д.

В зависимости от того, какие параметры были включены, производится сборка драйвера с нужными компонентами. Например, если производится сборка драйвера с поддержкой X11, то на выходе будут дополнительно собраны библиотеки драйвера с API OpenGL, 2D-драйвер для взаимодействия с X-системой, DRI-драйвер. В другом случае, например, при сборке FB-драйвера, эти компоненты не будут собраны, так как они не требуются для функционирования драйвера в данном режиме.

Взаимодействие с платформой Protium

Protium – платформа для создания прототипов от компании Cadence. Она позволяет использовать имеющееся описание микропроцессора на языке описания аппаратуры и создавать на его основе прототип на базе ПЛИС [5]. Подключение платформы прототипирования к рабочей машине происходит с помощью адаптеров SpeedBridge Ethernet. Данные адаптеры предоставляют интерфейсы взаимодействия для систем эмуляции Palladium и систем прототипирования Protium с использованием FPGA, с внешними системами, сетями и тестовым оборудованием и позволяют проектным командам загружать ОС, осуществлять передачу файлов и отображать графику и видео с использованием моделируемых систем [6]. Доступ к прошивкам и файловым системам осуществляется также через Ethernet.

Protium не имеет модуля вывода информации на монитор, поэтому для взаимодействия с эмулируемой системой используется подключение к инструментальной машине через COM-порт. При помощи утилиты minicom осуществляются удаленный доступ на инструментальную машину и управление платформой Protium. В зависимости от прошивки для Protium необходимо изменять скорость передачи данных COM-порта при помощи скриптов. Особенностью работы с Protium является скорость выполнения команд. В силу того, что система имеет низкую частоту работы (3–20 МГц), скорость выполнения получается очень низкой, к примеру, загрузка ОС Linux может длиться 12–18 минут [5].

Запуск драйвера на отладочной системе

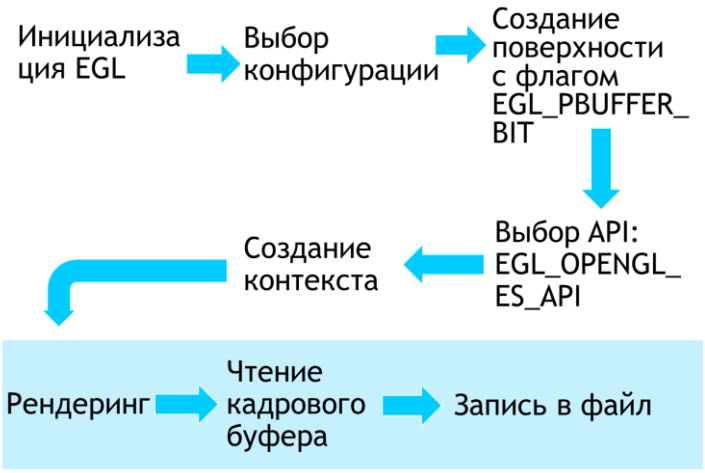
Для проверки функциональности собранного драйвера используется отладочная система – платформа прототипирования Protium. Собранный драйвер копируется в файловую систему, которая будет загружаться на отладочном стенде. После этого производится загрузка модуля драйвера. Как только модуль успешно загрузится в систему, можно приступать к тестированию драйвера. К примеру, если драйвер собирался с поддержкой FB, то запускаются соответствующие тестовые приложения. Если драйвер собирался с поддержкой X11, то сначала необходимо запустить X-систему. Для этого требуется определить конфигурации Xorg, то есть изменить файл xorg.conf, в котором нужно указать, какой драйвер будет использоваться X-системой. Далее запускается X-система. После успешного запуска можно приступить к тестированию драйверов.

Платформа прототипирования Protium позволяет моделировать системы с различным порядком байтов – big и little endian. Разрабатываемое под архитектуры little-endian ПО может не работать с big-endian, например, в графических приложениях могут возникнуть проблемы с загрузкой трехмерных моделей, текстур и т.д. Поэтому требуется адаптировать приложения для работы в системах с различным порядком байтов. Для этого на этапе загрузки модели или текстуры необходимо выполнять проверку на порядок байтов. В случае, если сборка программы выполнена под систему big-endian, а ресурсы созданы в little-endian, делается переворот байтов данных, например, для модели это будут координаты вершин, полигонов, а для текстур – значения цвета пикселей.

Получение результата тестирования графики

С учетом отсутствия модуля вывода информации на монитор для получения сгенерированного изображения необходимо сохранить кадры в файлы. Первый этап процедуры получения результата – создание кадрового буфера, в котором будет происходить рендеринг изображения (см. рисунок). Для этого необходимо выполнить инициализацию EGL (для OpenGL ES), предварительно получив подключение «дисплея» EGL. После успешной инициализации необходимо указать конфигурацию поверхности, в которой нужно указать следующие атрибуты: размеры компонентов цветового буфера в битах (красного, зеленого и синего), размер буфера глубины в битах, тип поверхности, тип API контекста. Отдельно задается конфигурация кадрового буфера, где указываются его размеры. Задав данные конфигурации, в приложении создается поверхность кадрового буфера, который будет использоваться при закадровом рендеринге. После успешного создания идет задание текущего API визуализации для EGL. Следующий шаг –

создание контекста, при этом необходимо указать в конфигурации версию контекста OpenGL ES. Заключительным шагом является связка контекста отображения EGL и поверхности. На этом первый этап завершается.



Процедура получения результата

Второй этап – сохранение сгенерированного кадра в файл, который выполняется во время рендеринга. Для этого необходимо получить данные из кадрового буфера. Сначала необходимо получить значения разрешения кадрового буфера, чтобы вычислить размер выделяемого буфера данных. После этого происходит считывание данных. Далее необходимо сформировать файл изображения, в который будут сохраняться данные. Осуществляется запись заголовка формата, например, для TGA записываются следующие параметры: заголовочные биты формата, положения по x, y, ширина, высота, глубина цвета. Затем данные записываются в созданный файл.

Тестирование OpenCL

Для тестирования вычислительных возможностей графического ядра использовался комплекс The OpenCL Conformance Tests. С его помощью можно проверить аппаратную поддержку OpenCL.

Для функционирования тестов необходима реализация режимов работы процессора, например, включение и отключение режима FTZ (Flush-To-Zero). Режим FTZ заменяет денормализованные числа на ноль, что в некоторых случаях может значительно повысить производительность [7]. Реализация данного режима будет отличаться для различных архитектур процессоров. Для архитектур ARM и MIPS данный режим реализуется с помощью ассемблерных вставок. К примеру, в MIPS используются команды mfc1 и mtc1. Для включения FTZ-режима осуществляется получение текущего значения режима командой mfc1 из 31 регистра. Также идет сохранение текущего режима. Для включения полученное значение изменяется операцией побитовое «ИЛИ» и заносится обратно в 31 регистр командой mtc1. Чтобы отключить режим для изменения значения бита, используется операция побитового «И». Также реализован сброс устройства до исходного состояния. Он выполняется занесением в 31 регистр сохраненного до этого значения режима командой mtc1.

Заключение

В статье дано общее описание графического стека в ОС Linux, компоненты которого разделяются по областям памяти пространства ядра и пространства задач. Рассмотрены инструментарии, с помощью которых производятся сборка и отладка драйверов, а также особенности кросс-компиляции под различные архитектуры. Описаны этапы подготовки, в том числе настройка параметров для графического ядра, рабочего окружения и сборки драйверов с различными конфигурациями. Также в статье описываются метод тестирования драйверов и получение результата на системе прототипирования Protium с учетом ее особенностей. К ним можно отнести в том числе и низкую скорость выполнения команд, отсутствие вывода на монитор. Приведен подход для тестирования функционала OpenCL с помощью специализированного комплекса тестов, который был доработан для корректной работы на целевой платформе. Планируется дальнейшая проработка методов отладки графических драйверов, в частности, на уровне ядра ОС.

Литература

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.

2. The Frame Buffer Device. URL: https://www.kernel.org/doc/Documentation/fb/framebuffer.txt. (дата обращения: 24.09.18).

3. EGL Overview. URL: https://www.khronos.org/egl. (дата обращения: 24.09.18).

4. Toolchains. URL: https://elinux.org/Toolchains. (дата обращения: 24.09.18).

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

6. Cadence Speedbridge. URL: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/speedbridge-adapters.html (дата обращения: 24.09.18).

7. Flush-to-zero mode. URL: http://www.keil.com/support/man/docs/armasm/armasm_pge1423648462321.htm (дата обращения: 24.09.18).