ВИЗУАЛИЗАЦИЯ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ В ОПЕРАЦИОННЫХ СИСТЕМАХ ОБЩЕГО НАЗНАЧЕНИЯ

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

Научно-исследовательский институт системных исследований РАН, https://www.niisi.ru/

Москва 117218, Российская Федерация

Поступила 23.01.2019, принята 05.02.2019

Представлена действительным членом РАЕН И.Б. Петровым

В статье описаны подходы к разработке программного обеспечения для задач взаимодействия контроллера вывода на экран и операционной системы (ОС) Linux. Архитектурой ОС предусмотрено создание драйвера — компонента, обеспечивающего корректное взаимодействие аппаратного контроллера с ядром системы с использованием различных протоколов. Разработка драйвера контроллеров затрудняется ввиду непрерывных изменений структуры ядра, не имеющих обратной совместимости. В данной статье рассмотрено несколько подходов к разработке драйверов контроллеров, основными из которых являются программирование режима контроллера внутри ядра Linux (Kernel Mode Setting, или KMS), а также программирование графического контроллера из графического сервера (User Mode Setting, или UMS). Рассмотрено историческое развитие подходов к написанию драйверов контроллера, а также причины принятия решений, которые в итоге привели к тому, что доминирующим подходом стало написание драйверов контроллера внутри ядра ОС. В ядре Linux применяется особый подход к написанию драйверов, который сочетает в себе процедурное и объектно-ориентированнное программирование путем использования свойств языка С и его расширений, используемых в компиляторе GCC. Из-за большого количества необходимых расширений сборка драйверов и ядра Linux затруднена любыми другими компиляторами. Новейшей концепцией написания драйверов контроллера вывода на экран для ядра Linux является концепция атомарных драйверов KMS. В современном Linux для контроллеров вывода на экран существует концепция состояний, которая позволяет отменять и применять состояния без промежуточных шагов. Для работы концепция состояний для режимов дисплея в ядре Linux требует реализации в соответствии с принципом «функция отмены полностью отменяет функцию включения, вплоть до состояния». Рассмотрены подходы к реализации драйвера контроллера вывода для ОС Linux, а также подробно рассмотрен способ, наиболее применяемый на текущий момент — модуль атомарного переключения режимов в пространстве ядра.

Ключевые слова: **драйвер, встраиваемые системы, Linux, модуль ядра, разработка**

УДК 004.454

Содержание

1. Введение (217)
2. Компоненты графической системы (218)
3. Графическая система и драйверы Linux (218)
4. Kernel mode setting и user mode setting. История вопроса (219)
5. Объектная модель ядра Linux (220)
6. Атомарный драйвер KMS. Основные сущности и необходимые к реализации интерфейсы (221)
7. Отладка драйвера (222)
8. Заключение (223)

Литература (223)

1. ВВЕДЕНИЕ

В настоящее время, в связи с увеличением доли различных носимых и встраиваемых систем, является актуальной задача разработки драйверов для них. На многих встраиваемых системах используется ОС Linux, которая имеет поддержку большого числа процессорных архитектур, а также поддержку загружаемых модулей, позволяющую сильно упростить процесс добавления драйвера.

Во многих встраиваемых системах на основе Linux вывод на экран информации

имеет свою специфику. Зачастую для конкретной встраиваемой системы требуется разработка специализированных драйверов для применяемого контроллера вывода.

При разработке драйвера контроллера вывода на основе ОС Linux необходимо учитывать ряд аспектов, связанных со стилем разработки ядра, паттернами разработки, быстрым изменением интерфейса взаимодействия с ядром (API). Документации по разработке драйверов контроллера вывода на экран в открытом доступе также достаточно мало, и, в основном, ограничиваются выписками из комментариев в файлах исходного кода ядра Linux, либо схемами на их основе. В данной статье рассмотрен современный подход для написания драйвера контроллера вывода на экран для ОС Linux.

2. КОМПОНЕНТЫ ГРАФИЧЕСКОЙ СИСТЕМЫ

В связи с необходимостью 3D-рендеринга, высокоинтенсивного расчета шейдеров, обработки видео большого разрешения и прочими потребностями современного пользователя компьютерных систем, графическая подсистема ПК по сравнению с ею же в ранние годы серьезно усложнилась, и чаще всего современная графическая система включает в себя [1]:

1. Графический ускоритель (GPU). Именно его производительность в основном измеряют тестами и различными экспериментальными средствами (к примеру, запуском 3D-приложений на максимальных настройках или специально созданных сцен), именно он занимается выполнением шейдеров при помощи специализированной многопоточной архитектуры.
2. Контроллер вывода на экран. Данная часть графической подсистемы используется для определения режимов вывода графики на экран, а также для взаимодействия с мониторами (получение доступных разрешений монитора, пересылка готового кадра и т.д.)
3. Также в графической системе можно найти специализированные чипы для видеовывода, для взаимодействия с устройствами захвата видео, и специализированную оперативную память (GDDR) для использования графическим ускорителем.

Данный список компонентов не является обязательным, так как существуют графические подсистемы, в которых один или несколько компонентов отсутствует, либо не задействованы. Например, в некоторых решениях NVIDIA Optimus, устанавливаемых в ноутбуки, не задействован контроллер вывода на экран, вся настройка режимов монитора происходят через контроллер другой графической карты – Intel. От NVIDIA используется только графический ускоритель. Также можно найти и системы с другим сочетанием выбранных компонентов графической системы.

3. ГРАФИЧЕСКАЯ СИСТЕМА И ДРАЙВЕРЫ LINUX

Графическая система в Linux представлена графическим стеком и использующимися в нём различными библиотеками (**Рис. 1**) [2].

Если рассматривать компоненты графического стека Linux сверху вниз, то можно сказать, что подавляющее большинство из них предназначено для взаимодействия с GPU, в частности: большая

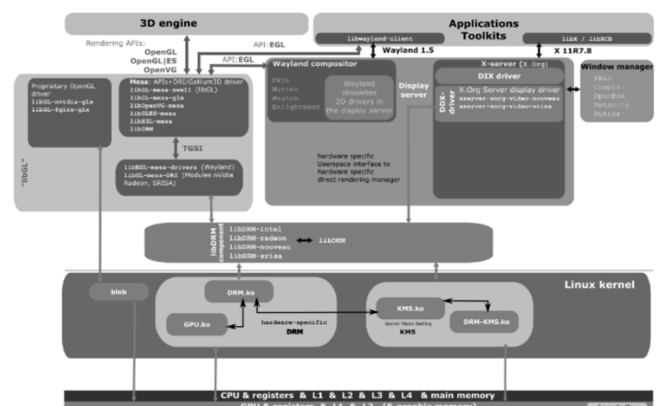

Рис. 1. *Графический стек Linux.*

часть Wayland и libdrm, полностью EGL и практически полностью X. С контроллером вывода на экран взаимодействует только небольшая часть DDX (Device Dependent X, Рис. 1.) и Wayland, а также libgbm.

4. KERNEL MODE SETTING И USER MODE SETTING. ИСТОРИЯ ВОПРОСА

Изначально существовал только user mode setting, который производился в драйверах DDX и позволял установить режим для отображения графической информации [3]. Установка часто производилась посредством таблиц режимов, и была нерасширяемой, либо производилась при помощи алгоритмов внутри кода, которые были уникальны для каждого чипа. Также имелось такое решение, как VBE (Vesa BIOS Extensions), расширение для IBM PC BIOS, которое позволяло осуществлять некоторые ограниченные операции установки режимов с его использованием (без написания прямого алгоритма для взаимодействия с графической подсистемой). Постепенно большинство драйверов перешло на VBE. Но у VBE имелось несколько серьезных недостатков:

- Отсутствие поддержки нескольких контроллеров CRT (в большинстве случаев необходимо для поддержки вывода с нескольких графических ускорителей на несколько дисплеев или вывода на TV).
- Проблемы, если имелось несколько разных выводов (и они могли быть разными на одной серии чипов).
- Проблемы с использованием нескольких разных таймеров.

Для решения данных проблем в 2008 году было принято в ядро расширение [4], которое дало поддержку настройки режимов внутри ядра (kernel mode setting, или KMS). Наряду с проблемами, которые вносило данное расширение (такими как некоторое уменьшение стабильности ядра вследствие не очень качественно написанных драйверов, а также разрастания ядра вследствие включения в него множества драйверов графики), также имелись и некоторые преимущества, такие как упрощение отладки (в связи с нахождением всего кода настройки режимов внутри ядра), упрощение управления питанием, и удаление старого кода драйверов кадрового буфера (в эпоху таблиц и VBE (а для некоторых драйверов и сейчас) в ядре содержалась также подсистема управления кадровым буфером, позволявшая использовать графический вывод без загрузки графического сервера для вывода сообщений консоли, либо для прорисовки консольных программ).

Первая версия протокола KMS [5] была построена по модели XRandR 1.2, которая была достаточно хороша на тот момент для установки режимов на обычных рабочих станциях. Но с появлением мобильных устройств возможностей первой спецификации KMS стало не хватать — с целью снижения энергопотребления была разработана концепция слоев (planes), которая использовалась для отображения видео на отдельном экране, для задействования аппаратного курсора (в первых версиях KMS аппаратный курсор взаимодействовал с контроллером режимов экрана по отдельному протоколу), и для других подобных случаев.

Затем добавилась поддержка других объектов KMS, которые позволяли изменить гамму отображаемого вывода, поворачивать выводимое изображение прямо в драйвере и прочее [6].

Проблема у всего этого осталась одна — для взаимодействия указанных подсистем с ядром требовались большое число ioctl. И каждое устройство при попытках взаимодействия должно было проверять каждый ioctl, что было очень неудобным.

В связи с данными проблемам KMS на мобильных устройствах компанией Google для ядра Linux в составе OS Android был

разработан собственный интерфейс для установки режимов — Android Atomic Display Framework (ADF) [7].

Несмотря на множество достоинств (атомарное обновление слоев, легкость разработки новых драйверов) ADF обладал следующими недостатками:

- Одна очередь обновления, и, в связи с этим, очень плохая поддержка многоэкранных режимов. В случае, если имелось 2 дисплея с разными частотами обновления, то частота обновления приводилась к одинаковой (чаще всего к меньшей), что давало либо рывки на более быстром дисплее, либо замедление на более медленном.
- ADF предполагал, что в пространстве пользователя имеется специфичный драйвер для графического чипа, обязательно предоставляющий 2D или 3D ускорение, что не устраивало разработчиков основного ядра, так как это отсекало все «общие» драйверы (предназначенные для пока не поддерживаемых устройств и предоставляющие минимальный функционал) типа xf86-video-modesetting.
- У ADF также были трудности с обновлением цепочки выводов. Поскольку в драйверах с поддержкой нескольких выводов очень много разделяемых ресурсов, требующих однократного обновления вне зависимости от количества выводов, тогда как у ADF было требование к атомарному обновлению всей цепочки одного вывода. Для нескольких выводов это можно было реализовать в цикле, но такая реализация не работала в случае нескольких выводов и разделяемых ресурсов.
- ADF использовал не рекомендуемый в разработке ядра паттерн «прослойка» (midlayer). Данный паттерн не используется, поскольку, несмотря на кажущееся удобство использования и удобство разработки драйверов под прослойку, всегда может найтись вендор, которого прослойка не устроит — это неприемлемо для ядра.
- ADF был новым, и отсутствовала обратная совместимость со старыми интерфейсами и API.

Поэтому было принято решение не сохранять ADF, но внедрить его лучшие черты в KMS [5]. Так в KMS появилась концепция атомарных обновлений. Современный драйвер контроллера вывода на экран обязательно должен быть написан именно в пространстве ядра с использованием данной концепции.

5. ОБЪЕКТНАЯ МОДЕЛЬ ЯДРА LINUX

Для написания драйвера в пространстве ядра необходимо использовать его объектную модель [8]. Несмотря на то, что ядро Linux написано на *C*, внутри него имеется объектная модель, которая позволяет применять некоторые подходы ООП при написании драйверов ядра. Для построения данной объектной модели чаще всего используются следующие способы:

- Установление взаимно-однозначного соответствия между структурами специального вида и объектами (в понимании ООП). Структура специального вида содержит в себе:
  - указатели на функции, либо указатель на структуру vtable (куда подключаются виртуальные методы);
  - указатель типа void*, куда подключается закрытая часть данных структуры (аналог закрытого типа наследования);
  - содержит в себе объект базового типа целиком (не в форме указателя) для обеспечения связности родителя и наследника через специальные команды компилятора.
- Использование специального наименования и сигнатур функций

(имя_типа_объекта_имя_операции (тип_объекта* объект, ...)), которое позволяет точно определять тип объекта, для которого предназначена данная функция. Можно установить взаимно-однозначное соответствие между методами в понимании ООП и такими функциями.

- Структуры vtable. *C* (в отличие от *C*++) не имеет такой сущности, как виртуальные функции, таблицы виртуальных функций и подобные им вещи. Поэтому разработчики ядра для хранения виртуальных методов используют структуры, состоящие только из указателей на функции, либо хранящие кроме таких указателей также переменные, которые подобны переменным защищенного типа наследования в ООП. Данные структуры можно сопоставить с vtable из *C*++.
- Для построения приватных полей в некоторых структурах имеются void* переменные, в которые можно присвоить закрытую структуру (в частности, при инициализации драйвера KMS необходимо создать закрытую структуру в структуре drm_device) container_of вместе с хранением структуры "базового класса" в "наследнике". Для того, чтобы упростить реализацию наследования в структурах ядра, часто используется такой способ, как хранение полной базовой структуры внутри производной, а при необходимости получения производной структуры по базовой используется макрос container_of, который внутри себя использует команду компилятора offsetof, позволяющий находить базовую структуру внутри производной.

Все эти вещи в целом позволяют существенно упростить написание кода для пространства ядра Linux и используются в большинстве имеющихся драйверов.

6. АТОМАРНЫЙ ДРАЙВЕР KMS. ОСНОВНЫЕ СУЩНОСТИ И НЕОБХОДИМЫЕ К РЕАЛИЗАЦИИ ИНТЕРФЕЙСЫ

В данном разделе будет рассмотрен вопрос создания модуля ядра Linux, который работает через DRM и KMS с контроллером дисплея и не задействует других компонентов графической подсистемы. Такой драйвер чаще всего пишется под встраиваемые системы, поскольку в них могут использоваться компоненты разных производителей (к примеру, 3D-чип от Mali, а контроллер вывода на экран — от Kirin), что необходимо учитывать при разработке драйвера каждого компонента.

Основные компоненты драйвера KMS (**Рис. 2**) [9]:

1. Объект драйвера (drm_driver). В тело структуры драйвера записываются общие параметры (вроде его версии, создателя и поддерживаемых интерфейсов), а также определяются общие вопросы работы драйвера (работа с GEM, прерываниями и файлами устройств). В отличие от общих практик написания объектов в ядре, объект драйвера содержит большинство указателей на виртуальные методы (в структуру vtable вынесены только операции с файлами). Изначально там определялись также работа с vblank, но была перенесена в CRTC (так как в случае нескольких CRTC у них может быть

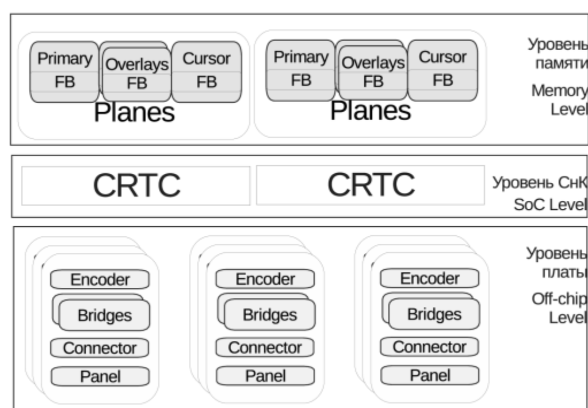

Рис. 2. *Пример основных компонентов драйвера KMS.*

разная работа с vblank).

2. Контроллер CRT (CRTC). В CRTC хранится состояние выставленного на дисплеях режима, объекты основного плана и курсора, атомарное состояние CRTC, а также бит включения. Функции, назначаемые CRTC — работа с vblank, со слоями (основным слоем, курсором и дополнительными слоями) и со страницами.

3. Один или несколько слоев. Для каждой CRTC должен быть ровно один основной слой, не более одного курсора, и сколько угодно дополнительных слоев. Слои привязываются к создавшей их CRTC.

4. Один или несколько кодировщиков (encoders). Кодировщики должны переводить сигнал из внутреннего формата DRM в формат вывода и наоборот, если требуется нетривиальный перевод. В нем корректируется режим, а также есть функции для управления энергопотреблением устройства вывода информации. Кодировщик жестко не привязывается к CRTC, а вместо этого в нем выставляются биты возможных для использования CRTC. Кодировщик также может проверять выставленный режим на допустимость, но обычно это используется только в нетривиальных случаях.

5. Коннектор. Коннектор определяет вывод и связывается с устройством напрямую и, через него, с одной или несколькими CRTC и кодировщиками. Также у коннектора есть состояние и именно в нем драйвер определяет проверку режима на то, возможно ли выставить его в данном выводе. Чаще всего количество коннекторов соответствует количеству видеовыводов. Во многих случаях коннектор сам занимается перекодировкой данных в нужный формат, в этом случае он связывается с пустым кодировщиком.

6. Мост (drm_bridge). Мост связывает несколько кодировщиков, если сигналу необходимо пройти несколько преобразований. По функционалу он подобен коннектору и кодировщику.

7. Панель (drm_panel). Панель представляет собой программное представление функций управления выводом и привязывается к коннектору. В ней выставляются функции для получения допустимых режимов и таймингов материнского коннектора.

7. ОТЛАДКА ДРАЙВЕРА

Для отладки драйвера пространства ядра [10] возможно использовать:

1. Ядро, собранное в Debug режиме (и интерфейсы такого ядра, выводимые через debugfs). Внутри ядра, собранного с поддержкой динамической отладки, существует виртуальная файловая система debugfs, в псевдофайлы которой выводится различная отладочная информация по запросу пользователя. Недостатками данной системы является то, что она работает только на загруженном ядре и необходимо читать файлы именно через отлаживаемый ПК, также данная система работает только с теми местами кода ядра, где посчитали нужным сами разработчики. Для произвольной отладки необходимо накладывать собственные отладочные изменения на ядро.

2. Отладчик KGDB. Отладчик KGDB — это GDB, запущенный на удаленном ПК, подключенном через интерфейс последовательного порта к отлаживаемой плате. Для этого внутри ядра есть возможность сборки с поддержкой ожидания подключения из интерфейса GDB, с поддержкой вывода сообщений об ошибках и трассировкой стека в последовательный порт. Со стороны удаленного ПК пользователя необходима установка отладчика GDB, а также

экземпляр ядра, собранные в режиме отладки с отладочной информацией, только в этом случае KGDB даст информацию о происходящем внутри ядра. Для разделения обычного вывода и вывода отладки используются специальные программы-микшеры, работающие как прослойка между отладчиком и ядром.

3. Специализированные FPGA, такие как, например, Palladium, которые позволяют проследить получение сигналов графической подсистемой [11]. Данные FPGA имеют доступ к сигналам и значениям в регистрах устройств, которые они эмулируют, что позволяет при отладке драйверов сверять значения ядра с записанными значениями.

8. ЗАКЛЮЧЕНИЕ

При разработке драйвера контроллера вывода на экран для встраиваемых систем необходимо учитывать ряд аспектов, таких как трудности отладки, особенная объектная модель ядра Linux, специфика взаимодействия с аппаратной частью. Поскольку контроллеры вывода на экран имеются практически в каждой встраиваемой системе (за исключением тех, которые не используют вывод на экран вообще), задача разработки драйверов контроллеров на них, подходов к их проектированию, отладке и тестированию в соответствии с объектной моделью ядра Linux является актуальной задачей. В настоящее время разработка данных драйверов ведется в основном закрытым путем в больших компаниях, которые выкладывают в открытый доступ только конечный результат, что не лучшим способом сказывается на развитии единого подхода к разработке драйверов контроллера вывода на экран и подходов к способам установки режимов отображения экрана вообще.

В данной статье рассмотрены подходы к написанию драйвера контроллера вывода режимов, применяемые в различное время в ОС Linux, а также детально рассмотрен новейших подход к разработке данных драйверов для встраиваемых систем. Практическая реализация драйвера такого типа может быть использована как с драйверами GPU, так и с новыми разработками в области программной растеризации (в частности, llvmpipe).

Благодарности

Публикация выполнена в рамках государственного задания по проведению фундаментальных научных исследований (ГП 14) по теме (проекту) № 0065-2019-0001.

ЛИТЕРАТУРА

1. Madieu J. *Linux Device Drivers Development*. Birmingham, Packt Publishing, 2017, 586 p.
2. Ефремов ИА, Мамросенко КА, Решетников ВН. Методы разработки драйверов графической подсистемы. *Программные продукты и системы*, 2018, 31(3):425-429.
3. Verhaegen L. *X and Modesetting: Atrophy illustrated*. 2006, p. 7.
4. Airlie D, Barnes J, Bornecrantz J. *DRM: add mode setting support*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f453ba0460742ad027ae0c4c7d61e62817b3e7ef (дата обращения: 04.09.2018).
5. Vetter D. *Atomic mode setting design overview, part 1*. URL: https://lwn.net/Articles/653071/ (дата обращения: 04.09.2018).
6. Pinchart L. Why and How to use KMS as Your Userspace Display API of Choice. *LinuxCon Japan Tokyo*, 2013, 52 p.
7. Syrjälä V, Clark R, Hackmann G. [RFC 0/4] *Atomic Display Framework*. URL: https://lists.freedesktop.org/archives/dri-devel/2013-August/044522.html (дата обращения: 20.11.2018).
8. Brown N. *Object-oriented design patterns in the kernel, part 1*. URL: https://lwn.net/Articles/444910/ (дата обращения: 04.09.2018).
9. The Linux Kernel Documentation. URL: https://www.kernel.org/doc/html/v4.11/gpu/drm-kms.html (дата обращения: 20.04.2018).

10. Баженов ПС, Мамросенко КА, Решетников ВН. Исследование и отладка компонентов для обработки трехмерной графики операционных систем. *Программные продукты, системы и алгоритмы*, 2018, 4:5.
11. Богданов АЮ. Опыт применения платформы прототипирования на ПЛИС «Protium» для верификации микропроцессоров. *Труды НИИСИ РАН*, 2017, 7(2):46-49.

Пугин Константин Витальевич
программист
НИИ Системных исследований РАН, Отделение разработки вычислительных систем Центра визуализации и спутниковых информационных технологий
36/1, Нахимовский просп., Москва 117218, Россия
rilian@niisi.ras.ru

Мамросенко Кирилл Анатольевич
к.т.н., руководитель Центра
НИИ Системных исследований РАН, Отделение разработки вычислительных систем Центра визуализации и спутниковых информационных технологий
36/1, Нахимовский просп., Москва 117218, Россия
mamrosenko_k@niisi.ras.ru

Гиацинтов Александр Михайлович
к.т.н., старший научный сотрудник
НИИ Системных исследований РАН, Отделение разработки вычислительных систем Центра визуализации и спутниковых информационных технологий
36/1, Нахимовский просп., Москва 117218, Россия
algts@niisi.ras.ru.

TECHNOLOGIES OF GRAPHICS INFORMATION PROCESSING AND OUTPUT IN GENERAL-PURPOSE OPERATING SYSTEMS

Konstantin V. Pugin, Kirill A. Mamrosenko, Alexander M. Giatsintov

Scientific Research Institute of System Analysis of the RAS, Center of Visualization and Satellite Information Technologies, https://niisi.ru/
Moscow 117218, Russian Federation
rilian@niisi.ras.ru, mamrosenko_k@niisi.ras.ru, algts@niisi.ras.ru

Abstract: **Article describes solutions for developing programs for interaction between Linux operating system and display controller. Operating system architecture encourages creating a driver — component, which task is to perform interaction of hardware controller and OS kernel with the use of many protocols. The development of drivers for the open source OS is difficult due to continuous changes in the structure of the kernel, which breaks backward compatibility frequently. Several approaches to display controller driver development are provided in the article. Basic concepts of these drivers include Kernel Mode Setting (or KMS), meant to provide driver in the kernel and User Mode Setting, meant to provide driver in graphical server. Specific way is used for develop Linux kernel drivers - it is half-procedural, and half-object oriented. Sometimes it is called as "OOP in C", because it is based on GNU C Language extensions usage to simulate object-oriented techniques. GNU C usage almost prevents using non-GNU compilers to build a driver. Newest concept of writing display controller driver for Linux kernel is based upon atomic mode setting concept, in modern Linux it is achieved via state concept - intermediate states of an object are stored and modified without side effects to other objects. In order to make this concept work, driver should conform to the principle "disable function undoes all effects of enable and nothing more". Several approaches for display driver development are provided in this article, and most modern method – atomic KMS mode setting, - is described in detail.**

Keywords: **drivers, embedded, KMS, kernel module, development**

UDC 004.454

Bibliography - 11 references *Received 23.01.2019, accepted 05.02.2019*

RENSIT, 2019, 11(2):217-224 **DOI: 10.17725/rensit.2019.11.217**