Модель драйвера ядра Linux

Patrick Mochel <mochel@digitalimplant.org>,
Tejun Heo <teheo@suse.de>,
Patrick Mochel

25 марта 2020

Аннотация

Сборник переводов статей из каталога *linux-4.4.190/Documentation/driver-model*.

Содержание

1 Модель устройства в ядре Linux - overview.txt **4**

1.1 Введение . . . 4

1.2 Нисходящий доступ . . . 4

1.3 Интерфейс пользователя . . . 5

2 Устройства и драйверы платформы - platform.txt **5**

2.1 Устройства платформы . . . 5

2.2 Драйверы платформы . . . 6

2.3 Перечисление устройств . . . 7

2.4 Устаревшие драйверы: проверка устройств . . . 7

2.5 Наименование устройств и привязка драйверов . . . 8

2.6 Ранние устройства и драйверы платформы . . . 8

2.6.1 Регистрация данных ранних устройства платформы . . . 9

2.6.2 Разбор параметров командной строки ядра . . . 9

2.6.3 Установка ранних драйверов платформы, принадлежащих к определенному классу . . . 9

2.6.4 Регистрация ранних драйверов платформы . . . 9

2.6.5 Зондирование (probing) ранних драйверов платформы, принадлежащих к определенному классу . . . 9

2.6.6 Содержимое функции probe() ранних драйверов платформы . . . 9

3 Классы устройств - class.txt **10**

3.1 Введение . . . 10

3.2 Программный интерфейс . . . 10

3.3 Устройства . . . 10
3.4 Драйверы устройств . . . 11
3.5 Структура каталога sysfs . . . 11
3.6 Экспортируемые атрибуты . . . 11
3.7 Интерфейсы . . . 12

4 Тип шин - bus.txt **12**

4.1 Регистрация . . . 12
4.2 Объявление . . . 12
4.3 Регистрация . . . 12
4.4 Функции обратного вызова . . . 13
4.5 match(): Связывание драйверов с устройствами . . . 13
4.6 Списки устройств и драйверов . . . 13
4.7 sysfs . . . 13
4.8 Экспорт атрибутов . . . 14

5 Драйверы устройств - driver.txt **14**

5.1 Размещение . . . 14
5.2 Инициализация . . . 14
5.3 Объявление . . . 15
5.4 Регистрация . . . 15
5.5 Драйверы промежуточных шин . . . 16
5.6 Доступ . . . 16
5.7 sysfs . . . 16
5.8 Функции обратного вызова . . . 16
5.9 Атрибуты . . . 17

6 Основы структуры устройства - device.txt **18**

6.1 Интерфейс программирования . . . 18
6.2 Атрибуты . . . 18

7 Паттерны проектирования драйверов устройств - design-patterns.txt **20**

7.1 State Container . . . 20
7.2 container_of() . . . 21

8 Привязка драйвера - binding.txt **22**

8.1 Шина . . . 22

8.2 device_register . . . 22

8.3 int match(struct device * dev, struct device_driver * drv); . . . 22

8.4 Класс устройства . . . 22

8.5 Драйвер . . . 22

8.6 sysfs . . . 22

8.7 driver_register . . . 23

8.8 Удаление . . . 23

9 Управление ресурсами устройства - devres.txt **23**

9.1 Ведение
А? Devres? . . . 23

9.2 Список Devres
Devres iв двух словах . . . 23

9.3 Группы в Devres
Сгруппируйте devres-ы и реализуйте их вместе . . . 25

9.4 Подробности
Правила времени жизни, контекст вызова, ... 26

9.5 Накладные расходы
Сколько мы должны заплатить за это? . . . 26

9.6 Список интерфейсов управления
В настоящее время реализованы интерфейсы управления . . . 26

10 Портирование существующего драйвера на новую модель - porting.txt **28**

10.1 Обзор . . . 28

10.2 Процесс портирования . . . 28

10.2.1 Step 0: Подготовка . . . 28

10.2.2 Step 1: Регистрация драйвера шины. . . . 28

10.2.3 Step 2: Регистрация устройства. . . . 29

10.2.4 Step 3: Registering Drivers. . . . 31

10.2.5 Step 4: Define Generic Methods for Drivers. . . . 32

10.2.6 Step 5: Support generic driver binding. . . . 32

10.2.7 Step 6: Supply a hotplug callback. . . . 33

10.2.8 Step 7: Cleaning up the bus driver. . . . 33

1 Модель устройства в ядре Linux - overview.txt

1.1 Введение

Модель драйвера ядра Linux - это объединение всех разнородных моделей драйверов, которые ранее использовались в ядре. Она предназначена для расширения драйверов шины для мостов и устройств путем объединения набора данных и операций в глобально доступные структуры данных.

Традиционные модели драйверов реализовали своего рода древовидную структуру (иногда просто список) для устройств, которыми они управляют. Не было никакой однородности между различными типами автобусов.

Текущая модель драйвера предоставляет общую унифицированную модель данных для описания шины и устройств, которые могут отображаться под шиной. Модель унифицированной шины включает в себя набор общих атрибутов, которые несут все шины, и набор общих обратных вызовов, таких как обнаружение устройства во время проверки шины, отключение шины, управление питанием шины и т.д.

Общий интерфейс устройства и моста отражает цели современного компьютера, а именно: возможность бесперебойной работы устройства «включай и работай», управление питанием и горячее подключение. В частности, модель, продиктованная Intel и Microsoft (а именно ACPI), гарантирует, что почти каждое устройство практически на любой шине в x86-совместимой системе может работать в рамках этой парадигмы. Конечно, не каждая шина способна поддерживать все такие операции, хотя большинство шин поддерживает большинство этих операций.

1.2 Нисходящий доступ

Общие поля данных были перемещены из отдельных слоев шины в общую структуру данных. Эти поля по-прежнему должны быть доступны для слоев шины, а иногда и для драйверов конкретного устройства.

Другим уровням шины рекомендуется делать то, что было сделано для уровня PCI. Структура *struct pci_ dev* теперь выглядит так:

Листинг 1: Структура pci_dev

```
1  struct pci_dev {
2          ...
3
4          struct device dev;      /* Generic device interface */
5          ...
6  };
```

Во-первых, обратите внимание, что устройство *struct device dev* внутри *struct pci_ dev* размещено статически. Это означает только одно выделение при обнаружении устройства.

Также обратите внимание, что это устройство *dev* для устройства не обязательно определяется в начале структуры *pci_ dev*. Это сделано для того, чтобы люди думали о том, что они делают, когда переключаются между драйвером шины и глобальным драйвером, и не поощряют бессмысленные и неправильные преобразования между ними.

Уровень шины PCI свободно обращается к полям *struct device*. Он знает о структуре *struct pci_ dev* и должен знать структуру *struct device*. Отдельные драйверы устройств PCI, которые были преобразованы в текущую модель драйверов, обычно не затрагивают и не должны затрагивать поля *struct device*, если для этого нет веских причин.

Вышеупомянутая абстракция избавляет от ненужной головной боли во время переходных фаз. Если бы это не было сделано таким образом, тогда, когда поле было переименовано или удалено, каждый последующий драйвер сломался бы. С другой стороны, если только слой шины (а не уровень устройства) получает доступ к устройству структуры, только уровень шины должен измениться.

1.3 Интерфейс пользователя

Благодаря полному иерархическому представлению всех устройств в системе, экспорт полного иерархического представления в пространство пользователя становится относительно простым. Это было достигнуто путем реализации виртуальной файловой системы специального назначения под названием *sysfs*.

Почти все основные дистрибутивы Linux монтируют эту файловую систему автоматически; вы можете увидеть некоторые варианты ниже в выводе команды «mount»:

Листинг 2: Выдача команды mount

```
$ mount
...
none on /sys type sysfs (rw,noexec,nosuid,nodev)
...
$
```

Автоматическое монтирование *sysfs* обычно выполняется с помощью строки в файле */etc/fstab*, аналогичной следующей:

Листинг 3: Строка таблицы fstab

```
none                /sys     sysfs    defaults               0 0
```

Если *sysfs* не монтируется автоматически, вы всегда можете сделать это вручную с помощью:

Листинг 4: Монтирование sysfs вручную

```
# mount -t sysfs sysfs /sys
```

Всякий раз, когда устройство вставляется в дерево, для него создается каталог. Этот каталог может быть заполнен на каждом уровне обнаружения - глобальном уровне, уровне шины или уровне устройства.

Глобальный слой в настоящее время создает два файла - «name» и «power». Первый сообщает только название устройства. Последний сообщает о текущем состоянии питания устройства. Он также будет использоваться для установки текущего состоянЪя питания.

Уровень шины также может создавать файлы для устройств, которые он обнаруживает во время проверки шины. Например, уровень PCI в настоящее время создает файлы «irq» и «resource» для каждого устройства PCI.

Драйвер, зависящий от устройства, может также экспортировать файлы в своем каталоге, или настраиваемые интерфейсы, чтобы предоставить данные, специфичные для устройства.

Дополнительную информацию о структуре каталога *sysfs* можно найти в других документах этого каталога и в файле *Documentation/filesystems/sysfs.txt*.

2 Устройства и драйверы платформы - platform.txt

Смотрите *<linux/platform_device.h>* для уточнения деталей интерфейса модели драйвера с шиной платформы: *platform_device* и *platform_driver*. Эта псевдо-шина используется для подключения устройств на шинах с минимальной инфраструктурой, например, используемых для интеграции периферийных устройств на многих процессорах системы на кристалле (SoC), или некоторых «устаревших» межсоединений ПК; в отличие от больших формально определенных, таких как PCI или USB.

2.1 Устройства платформы

Платформенные устройства - это устройства, которые обычно отображаются как автономные объекты в системе. Сюда входят устаревшие устройства на основе портов и хост-мосты с периферийными шинами, а также

большинство контроллеров, интегрированных в платформы «система-на-кристалле» (SoC). Их обычно объединяет прямая адресация с шины процессора. Редко, платформа_устройство будет подключено через сегмент какого-либо другого типа шины; но его регистры будут по-прежнему иметь прямую адресацию.

Устройствам платформы дают имя, используемое в привязке драйверов, и выделяют список ресурсов, таких как адреса и IRQ.

Листинг 5: Структура platform_device

```
struct platform_device {
        const char      *name;
        u32             id;
        struct device   dev;
        u32             num_resources;
        struct resource *resource;
};
```

2.2 Драйверы платформы

Драйверы платформы следуют стандартному соглашению модели драйверов, где обнаружение / перечисление обрабатываются вне драйверов, а драйверы предоставляют методы *probe()* и *remove()*. Они поддерживают управление питанием и уведомления о завершении работы с использованием стандартных соглашений.

Листинг 6: Структура platform_driver

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*suspend_late)(struct platform_device *, pm_message_t state);
        int (*resume_early)(struct platform_device *);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
};
```

Обратите внимание, что функция *probe()* должна в целом проверять, что указанное аппаратное устройство действительно существует; иногда код установки платформы не может быть в этом уверен. *probe()* может использовать для этого ресурсы устройства, включая часы и платформу устройства.

Драйверы платформы регистрируют себя обычным способом:

Листинг 7: Регистрация драйвера платформы

```
int platform_driver_register(struct platform_driver *drv);
```

Или, в обычных ситуациях, когда известно, что устройство не поддерживает горячее подключение, подпрограмма *probe()* может находиться в разделе *init*, чтобы уменьшить объем используемой памяти во время выполнения драйвера:

Листинг 8: Функция probe платформы

```
int platform_driver_probe(struct platform_driver *drv,
int (*probe)(struct platform_device *))
```

Модули ядра могут состоять из нескольких драйверов платформы. Ядро платформы предоставляет помощников для регистрации и отмены регистрации массива драйверов:

Листинг 9: Регистрация нескольких драйверов платформы

```
int __platform_register_drivers(struct platform_driver * const *drivers,
```

```
2 unsigned int count, struct module *owner);
3 void platform_unregister_drivers(struct platform_driver * const *drivers,
4 unsigned int count);
```

Если при регистрации одного из драйверов происходит ошибка, все драйверы, зарегистрированные до этого момента, будут разрегистрированы в обратном порядке. Обратите внимание, что есть вспомогательный макрос, для регистрации драйверов, который передает *THIS_MODULE* в качестве параметра, задающего владельца:

Листинг 10: Вспомогательный макром регистрация нескольких драйверов

```
1 #define platform_register_driver(drivers, count)
```

2.3 Перечисление устройств

Как правило код установки, специфичен для платформы (и часто для платы) и он регистрирует устройства платформы:

Листинг 11: Регистрация устройств платформы

```
1 int platform_device_register(struct platform_device *pdev);
2 
3 int platform_add_devices(struct platform_device **pdevs, int ndev);
```

Общее правило - регистрировать только те устройства, которые действительно существуют, но в некоторых случаях могут быть зарегистрированы дополнительные устройства. Например, ядро может быть настроено для работы с внешним сетевым адаптером, который может быть установлен не на всех платах, или аналогичным образом для работы со встроенным контроллером, который некоторые платы могут не подключать к каким-либо периферийным устройствам.

В некоторых случаях загрузочная прошивка экспортирует таблицы с описанием устройств, которые установлены на данной плате. Без таких таблиц часто единственный способ для кода установки системы настроить правильные устройства - это собрать ядро для конкретной целевой платы. Такие специфичные для платы ядра являются общими для разработки встраиваемых и пользовательских систем.

Во многих случаях ресурсов памяти и IRQ, связанных с устройством платформы, недостаточно для работы драйвера устройства. Код установки платы часто предоставляет дополнительную информацию, используя поле *platform_data* устройства для хранения дополнительной информации.

Встраиваемые системы часто нуждаются в одних или нескольких часах для устройств платформы, которые обычно отключаются до тех пор, пока не потребуется их активность (для экономии энергии). Настройка системы также связывает эти часы с устройством, так что вызовы *clk_get(&pdev->dev, clock_name)* возвращают их по мере необходимости.

2.4 Устаревшие драйверы: проверка устройств

Некоторые драйверы не полностью соответствуют новой модели драйвера, потому что они выполняют некую роль, не связанную напрямую с драйвером: драйвер регистрирует свое платформенное устройство, а не оставляет его для системной инфраструктуры. Такие драйверы нельзя подключать в горячем или холодном режиме, так как эти механизмы требуют, чтобы устройство создавалось в другом системном компоненте, а не в драйвере.

Единственная «хорошая» причина для этого заключается в работе с проектами старых систем, которые, как и в случае с первоначальными компьютерами IBM, полагаются на подверженные ошибкам модели «зондирования оборудования» для определения конфигурации оборудования. Более новые системы в значительной степени отказались от этой модели в пользу поддержки на уровне шины динамической конфигурации (PCI, USB) или таблиц устройств, предоставляемых загрузочной микропрограммой (например, PNPACPI на x86). Есть слишком много противоречивых вариантов того, что может быть, и даже обоснованные предположения операционной системы могут достаточно часто ошибаться, чтобы это создавало проблемы.

Этот стиль написания драйвера не рекомендуется. Если вы обновляете такой драйвер, попробуйте переместить перечисление устройств в более подходящее место вне драйвера. Обычно это будет очистка, поскольку такие драйверы, как правило, уже имеют «нормальные» режимы, например, те, которые используют узлы устройства, которые были созданы PNP или настройкой устройства платформы.

Тем не менее, есть кое-какое API для поддержки таких устаревших драйверов. Избегайте использования этих вызовов, кроме как для работы с драйверами, которые не поддерживают горячую замену.

Листинг 12: Динамическое создание устройства

```
1 struct platform_device *platform_device_alloc(
2 const char *name, int id);
```

Вы можете использовать *platform_device_alloc()* для динамического выделения устройства, которое вы затем инициализируете с помощью ресурсов и *platform_device_register()*. Лучшее решение обычно выглядит так:

Листинг 13: Динамическое создание устройства

```
1 struct platform_device *platform_device_register_simple(const char *name, int id,
2 struct resource *res, unsigned int nres);
```

Вы можете использовать вызов *platform_device_register_simple()* для выделения и регистрации устройства за один шаг.

2.5 Наименование устройств и привязка драйверов

platform_device.dev.bus_id - это каноническое имя для устройств. Оно состоит из двух компонентов:

- *platform_device.name* - используется для нахождения соответствующего драйвера.
- *platform_device.id* - номер экземпляра устройства, или 1"для указания, что устройство единственное.

Они объединены, поэтому *name/id "serial"/0* задаёт значение *bus_id* как *"serial.0"*, а *"serial/3"* задаёт значение *bus_id* как *"serial.3"*; оба будут использовать *platform_driver* с именем "serial". В то время как *"my_rtc"/-1* будет задавать *bus_id* как *"my_rtc"* (без идентификатора экземпляра) и использовать *platform_driver* с именем "my_rtc".

Привязка драйвера выполняется ядром автоматически, с помощью вызова *probe()* после обнаружения соответствия между устройством и драйвером. Если вызов *probe()* успешен, драйвер и устройство связываются как обычно. Есть три разных способа найти такое соответствие:

- Всякий раз, когда устройство зарегистрировано, драйверы для этой шины проверяются на совпадения. Устройства платформы должны быть зарегистрированы очень рано во время загрузки системы.
- Когда драйвер регистрируется с помощью *platform_driver_register()*, все несвязанные устройства на этой шине проверяются на совпадения. Драйверы обычно регистрируются позже во время загрузки ОС или после загрузки модуля драйвера.
- Регистрация драйвера с помощью *platform_driver_probe()* работает так же, как при использовании *platform_drive* за исключением того, что драйвер не будет проверяться позже, если зарегистрируется другое устройство. (Это нормально, поскольку этот интерфейс предназначен только для устройств, не поддерживающих горячее подключение.)

2.6 Ранние устройства и драйверы платформы

Ранние интерфейсы платформы предоставляют данные платформы драйверам устройств платформы на ранних этапах загрузки системы. Код построен поверх анализа командной строки *early_param()* и может быть выполнен очень рано.

Пример: "*earlyprintk*"класс ранней последовательной консоли работает в 6 шагов

2.6.1 Регистрация данных ранних устройства платформы

Код архитектуры регистрирует данные устройства платформы с помощью функции *early_platform_add_devices()*. В случае ранней последовательной консоли это должна быть аппаратная конфигурация для последовательного порта. Устройства, зарегистрированные в этот момент, будут позже сопоставлены с ранними драйверами платформы.

2.6.2 Разбор параметров командной строки ядра

Код архитектуры вызывает *parse_early_param()* для анализа командной строки ядра. Эта функция выполнит все соответствующие обратные вызовы *early_param()*. Указанные ранее ранние устройства платформы на этом этапе будут зарегистрированы. В случае ранней последовательной консоли пользователь может указать порт в командной строке ядра как "*earlyprintk = serial.0* где *earlyprintk* - строка класса, *serial* - имя драйвера платформы, а 0 - идентификатор устройства платформы. , Если идентификатор равен -1, то точка и идентификатор могут быть опущены.

2.6.3 Установка ранних драйверов платформы, принадлежащих к определенному классу

Код архитектуры может опционально принудительно регистрировать все ранние драйверы платформы, принадлежащие к определенному классу, используя функцию *early_platform_driver_register_all()*. Устройства, указанные пользователем на шаге 2, имеют приоритет над ними. Этот шаг пропущен в примере с драйвером последовательного порта, так как код раннего драйвера последовательного порта должен быть отключен, если пользователь не указал порт в командной строке ядра.

2.6.4 Регистрация ранних драйверов платформы

Скомпилированные драйверы платформы, использующие *early_platform_init()*, автоматически регистрируются на шаге 2 или 3. В примере с последовательным драйвером следует использовать *early_platform_init("earlyprint*

2.6.5 Зондирование (probing) ранних драйверов платформы, принадлежащих к определенному классу

Код архитектуры вызывает *early_platform_driver_probe()* для сопоставления зарегистрированных ранних платформенных устройств, связанных с определенным классом, с зарегистрированными ранними драйверами платформы. Для соответствующих устройства будут вызваны функции *probe()*. Этот шаг может быть выполнен в любой момент во время ранней загрузки. Было бы хорошо как можно скорее сделать это для случая последовательного порта.

2.6.6 Содержимое функции probe() ранних драйверов платформы

Код драйвера должен проявлять особую осторожность при ранней загрузке, особенно когда речь идет о распределении памяти и регистрации прерываний. Код в функции probe() может использовать *is_early_platform_device()*, чтобы проверить, вызывается ли он на раннем платформенном устройстве или во время обычного платформенного устройства. Ранний последовательный драйвер выполняет *register_console()* на этом этапе.

За более подробной информацией обращайтесь в <linux/platform_device.h>.

3 Классы устройств - class.txt

3.1 Введение

Класс устройства описывает тип устройства, например аудио или сетевое устройство. Были определены следующие классы устройств:

<Insert List of Device Classes Here>

Каждый класс устройств определяет набор семантики и программный интерфейс, которого придерживаются устройства этого класса. Драйверы устройств являются реализация этого интерфейса программирования для конкретного устройства на определенной шине.

Классы устройств не зависят от того, на какой шине находится устройство.

3.2 Программный интерфейс

Структура класса устройства выглядит как-то так:

Листинг 14: Методы класса устройства

```
1 typedef int (*devclass_add)(struct device *);
2 typedef void (*devclass_remove)(struct device *);
```

Подробно структура описана в kerneldoc.

Типичное определение класса устройства будет выглядеть так:

Листинг 15: Определение класса устройства

```
1 struct device_class input_devclass = {
2         .name           = "input",
3         .add_device     = input_add_device,
4         .remove_device  = input_remove_device,
5 };
```

Каждая структура класса устройства должна быть экспортирована в заголовочный файл, чтобы ее могли использовать драйверы, расширения и интерфейсы.

Классы устройств регистрируются и разрегистрируются в ядре с использованием:

Листинг 16: Регистрация классов устройства

```
1 int devclass_register(struct device_class * cls);
2 void devclass_unregister(struct device_class * cls);
```

3.3 Устройства

Поскольку устройства связаны с драйверами, они добавляются в класс устройств, к которому принадлежит драйвер. До модели драйвера ядра это обычно происходило во время обратного вызова *probe()* драйвера после инициализации устройства. Теперь это происходит после завершения обратного вызова *probe()* из ядра.

Устройства перечисляются в классе. Каждый раз, когда устройство добавляется в класс, поле *devnum* класса увеличивается и присваивается устройству. Поле никогда не уменьшается, поэтому, если устройство будет удалено из класса и добавлено повторно, оно получит другое перечисляемое значение.

Классу разрешено создавать специфичную для класса структуру для устройства и сохранять ее в указателе *class_data* устройства.

В классе устройств нет списка устройств. У каждого драйвера есть список устройств, которые он поддерживает. Класс устройства имеет список драйверов этого конкретного класса. Чтобы получить доступ ко всем устройствам в классе, переберите списки устройств каждого драйвера в классе.

3.4 Драйверы устройств

Драйверы устройств добавляются в классы устройств, когда они зарегистрированы в ядре. Драйвер определяет класс, к которому он принадлежит, устанавливая поле *struct device_ driver :: devclass*.

3.5 Структура каталога sysfs

В каталоге *sys* типа *sysfs* существует подкаталог с именем *class*.

Каждый класс получает каталог в каталоге классов вместе с двумя подкаталогами по умолчанию:

```
class/
`-- input
    |-- devices
    `-- drivers
```

Драйверы, зарегистрированные в классе, получают символическую ссылку в каталоге *drivers/*, которая указывает на каталог драйвера (в его каталоге *bus*):

```
class/
`-- input
    |-- devices
    `-- drivers
        `-- usb:usb_mouse -> ../../../bus/drivers/usb_mouse/
```

Каждое устройство получает символическую ссылку в каталоге *devices/*, которая указывает на каталог устройства в физической иерархии:

```
class/
`-- input
    |-- devices
    |   `-- 1 -> ../../../root/pci0/00:1f.0/usb_bus/00:1f.2-1:0/
    `-- drivers
```

3.6 Экспортируемые атрибуты

Листинг 17: Экспортируемые атрибуты

```
1 struct devclass_attribute {
2         struct attribute        attr;
3         ssize_t (*show)(struct device_class *, char * buf, size_t count, loff_t off);
4         ssize_t (*store)(struct device_class *, const char * buf, size_t count,
     loff_t off);
5 };
```

Драйверы класса могут экспортировать атрибуты, используя макрос *DEVCLASS_ ATTR*, который работает аналогично макросу *DEVICE_ ATTR* для устройств. Например, определение, подобное этому:

Листинг 18: Определение атрибута

```
1 static DEVCLASS_ATTR(debug,0644,show_debug,store_debug);
```

эквивалентно объявлению:

Листинг 19: Эквивалентное определение

```
static devclass_attribute devclass_attr_debug;
```

Драйвер шины может добавить и удалить атрибут из каталога *sysfs* класса, используя:

Листинг 20: Создание/удаление файлов атрибутов классов

```
int devclass_create_file(struct device_class *, struct devclass_attribute *);
void devclass_remove_file(struct device_class *, struct devclass_attribute *);
```

В приведенном выше примере файл будет называться «debug» и помещен в каталог класса в *sysfs*.

3.7 Интерфейсы

Может существовать несколько механизмов для доступа к одному и тому же устройству определенного типа класса. Интерфейсы устройств описывают эти механизмы.

Когда устройство добавляется в класс устройств, ядро пытается добавить его к каждому интерфейсу, зарегистрированному в классе устройств.

4 Тип шин - bus.txt

4.1 Регистрация

Смотрите *kerneldoc* для получения подробной подробной информации о структуре *bus_ type*.

Листинг 21: Регистрация шины

```
int bus_register(struct bus_type * bus);
```

4.2 Объявление

Каждый тип шины в ядре (PCI, USB и т. Д.) Должен объявлять один статический объект этого типа. Они должны инициализировать поле имени и могут дополнительно инициализировать соответствующий обратный вызов.

Листинг 22: Структура $bus_{t}ype$

```
struct bus_type pci_bus_type = {
        .name   = "pci",
        .match  = pci_bus_match,
};
```

Структура должна быть экспортирована в драйверы в заголовочном файле:

```
extern struct bus_type pci_bus_type;
```

4.3 Регистрация

Когда драйвер шины инициализируется, он вызывает $bus_{r}egister..,$.

4.4 Функции обратного вызова

4.5 match(): Связывание драйверов с устройствами

Формат структур Id устройств и семантика их сравнения по своей сути зависят от шины. Драйверы обычно объявляют массив Id устройств, которые они поддерживают, находящийся в структуре драйвера, специфичного для этой шины.

Назначение функции обратного обратного вызова состоит в том, чтобы дать шине возможность определить, поддерживает ли конкретный драйвер конкретное устройство, сравнивая идентификаторы устройств, поддерживаемые драйвером, с идентификатором данного конкретного устройства, не жертвуя при этом функциональными возможностями шины или безопасностью типов.

Когда драйвер зарегистрирован на шине, список устройств шины перебирается, и функция обратного вызова сопоставления вызывается для каждого устройства, с которым пока ещё не связан этот драйвер.

4.6 Списки устройств и драйверов

Списки устройств и драйверов предназначены для замены локальных списков, которые хранятся для многих шин. Это списки структур *devices* для устройств и *device_ drivers* для драйверов, соответственно. Драйвера шин могут использовать списки по своему усмотрению, но может потребоваться преобразование в конкретный тип шины.

Ядро LDM предоставляет вспомогательные функции для перебора каждого списка.

Листинг 23: Вспомогательбные функции для перебора списков

```
1 int bus_for_each_dev(struct bus_type * bus, struct device * start, void * data,
2 int (*fn)(struct device *, void *));
3
4 int bus_for_each_drv(struct bus_type * bus, struct device_driver * start,
5 void * data, int (*fn)(struct device_driver *, void *));
```

Эти вспомогательные функции перебирают соответствующий список и вызывают функции обратного вызова для каждого устройства или драйвера в списке. Все обращения к списку синхронизируются путем блокировки шины (читается текущее значение). Счетчик ссылок на каждый объект в списке увеличивается перед вызовом функции обратного вызова; он уменьшается после захвата следующего объекта. Блокировка не сохраняется во время вызове функции обратного вызова.

4.7 sysfs

Существует каталог верхнего уровня с именем «bus».

Каждой шине выделяется каталог в каталоге *bus* вместе с двумя подкаталогами по умолчанию:

```
/sys/bus/pci/
|-- devices
`-- drivers
```

Драйверы, зарегистрированные на шине, получают каталог в каталоге драйверов:

```
/sys/bus/pci/
|-- devices
`-- drivers
|-- Intel ICH
|-- Intel ICH Joystick
|-- agpgart
`-- e100
```

Каждое устройство, обнаруженное на шине такого типа, получает символическую ссылку в каталоге устройств шины на каталог устройства в физической иерархии:

```
/sys/bus/pci/
|-- devices
|   |-- 00:00.0 -> ../../../root/pci0/00:00.0
|   |-- 00:01.0 -> ../../../root/pci0/00:01.0
|   `-- 00:02.0 -> ../../../root/pci0/00:02.0
`-- drivers
```

4.8 Экспорт атрибутов

Листинг 24: Экспорт атрибутов

```
1 struct bus_attribute {
2         struct attribute        attr;
3         ssize_t (*show)(struct bus_type *, char * buf);
4         ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
5 };
```

Драйверы шины могут экспортировать атрибуты с помощью макроса BUS$_ATTR, DEVICE_ATTR.$, , :

```
1 static BUS_ATTR(debug,0644,show_debug,store_debug);
```

эквивалентно объявлению:

```
1 static bus_attribute bus_attr_debug;
```

Затем это можно использовать для добавления и удаления атрибута из каталога шины sysfs, используя:

Листинг 25: Создание файла в каталоге шины

```
1 int bus_create_file(struct bus_type *, struct bus_attribute *);
2 void bus_remove_file(struct bus_type *, struct bus_attribute *);
```

5 Драйверы устройств - driver.txt

Смотрите *kerneldoc* для получения подробной подробной информации о структуре *device_ driver*.

5.1 Размещение

Драйверы устройств являются статически распределенными структурами. Хотя в системе может быть несколько устройств, которые поддерживает этот драйвер, *struct device_ driver* представляет драйвер в целом (а не конкретный экземпляр устройства).

5.2 Инициализация

Драйвер должен инициализировать поля хотя бы name и bus. Он также должен инициализировать поле *devclass* (когда оно появится), чтобы он мог получить правильную связь внутри. Он также должен инициализировать как можно больше указателей на функции обратных вызовов, хотя все они являются необязательными.

5.3 Объявление

Как указано выше, объекты *struct device_ driver* размещаются статически. Ниже приведен пример объявления драйвера eepro100. Эта декларация является только гипотетической; она рассчитана на драйвер, полностью преобразованный в новую модель.

Листинг 26: Использовние структуры device$_{d}river$

```
static struct device_driver eepro100_driver = {
       .name             = "eepro100",
       .bus              = &pci_bus_type,

       .probe            = eepro100_probe,
       .remove           = eepro100_remove,
       .suspend          = eepro100_suspend,
       .resume           = eepro100_resume,
};
```

Большинство драйверов не смогут быть полностью преобразованы в новую модель, потому что шина, к которой они принадлежат, имеет специфическую для шины структуру с полями, специфичными именно для этой шины, которые не могут быть обобщены.

Наиболее распространенным примером этого являются структуры идентификаторов устройств. Драйвер обычно определяет массив идентификаторов устройств, которые он поддерживает. Формат этих структур и семантика для сравнения идентификаторов устройств полностью зависят от шины. Определение их как специфичных для шины объектов принесло бы в жертву безопасность типов, поэтому мы сохраняем специфичные для шины структуры.

Драйверы, специфичные для шины, должны включать общую структуру *struct device_ driver* в определение драйвера, специфичного для шины. Как здесь:

```
struct pci_driver {
       const struct pci_device_id *id_table;
       struct device_driver       driver;
};
```

Определение, включающее в себя специфичные для шины поля, будет выглядеть (снова используя драйвер eepro100):

Листинг 27: Использование структуры bpci_driver

```
static struct pci_driver eepro100_driver = {
       .id_table       = eepro100_pci_tbl,
       .driver         = {
                .name             = "eepro100",
                .bus              = &pci_bus_type,
                .probe            = eepro100_probe,
                .remove           = eepro100_remove,
                .suspend          = eepro100_suspend,
                .resume           = eepro100_resume,
       },
};
```

Некоторые могут найти синтаксис инициализации встроенной структуры неудобным или даже немного уродливым. Пока что это лучший способ сделать то, что мы хотим ...

5.4 Регистрация

```
int driver_register(struct device_driver * drv);
```

Драйвер регистрирует структуру при запуске. Для драйверов, которые не имеют специфичных для шины полей (то есть не имеют специфичной для шины структуры *driver*), будут использовать *driver_register* и передавать указатель на свой объект *struct device_driver*.

Однако большинство драйверов имеют специфичную для шины структуру и должны будут регистрироваться на шине, используя что-то вроде *pci_driver_register*.

Важно, чтобы драйверы регистрировали свою структуру драйверов как можно раньше. Регистрация в ядре инициализирует несколько полей в объекте *struct device_driver*, включая счетчик ссылок и блокировку. Предполагается, что эти поля действительны всегда и могут использоваться ядром модели устройства или драйвером шины.

5.5 Драйверы промежуточных шин

Определив функции-оболочки, можно упростить переход к новой модели. Драйверы могут полностью игнорировать общую структуру и позволить оболочке шины заполнять поля. Для обратных вызовов шина может определять общие обратные вызовы, которые переадресуют вызов специфичным для шины обратным вызовам драйверов.

Это решение должно быть только временным. Чтобы получить информацию о классе в драйвере, драйверы должны быть изменены в любом случае. Поскольку преобразование драйверов в новую модель должно уменьшить некоторую сложность инфраструктуры и размер кода, рекомендуется их преобразовывать по мере добавления информации о классе.

5.6 Доступ

Как только объект зарегистрирован, можно получить доступ к общим полям объекта, таким как блокировка и список устройств.

Листинг 28: Получение доступа к полям объекта

```
1 int driver_for_each_dev(struct device_driver * drv, void * data,
2                         int (*callback)(struct device * dev, void * data));
```

Поле *device* представляет собой список всех устройств, которые были связаны с драйвером. Ядро LDM предоставляет вспомогательную функцию для работы со всеми устройствами, которыми управляет драйвер. Эта функция блокирует драйвер при каждом доступе к узлу и правильно подсчитывает количество ссылок на каждом устройство, когда она обращается к нему.

5.7 sysfs

Когда драйвер зарегистрирован, в каталоге его шины создается каталог *sysfs*. В этом каталоге драйвер может экспортировать интерфейс в пользовательское пространство для управления работой драйвера в глобальном масштабе; например переключение вывода отладочной информации в драйвер.

В будущем функция этого каталога будет каталогом «устройств». Этот каталог будет содержать символические ссылки на каталоги устройств, которые он поддерживает.

5.8 Функции обратного вызова

Листинг 29: Функция probe

```
1 int     (*probe)        (struct device * dev);
```

Точка входа *probe()* вызывается в контексте задачи, когда семафор шины *rwsem* заблокирован, а драйвер частично привязан к устройству. Драйверы обычно используют *container_of()* для преобразования *dev* в

тип шины, как в *probe()*, так и в других подпрограммах. Этот тип часто предоставляет данные о ресурсах устройства, такие как *pci_dev.resource[]* или *platform_device.resources*, которые используются в дополнение к *dev->platform_data* для инициализации драйвера.

Этот обратный вызов содержит специфичную для драйвера логику для привязки драйвера к данному устройству. Она включает в себя проверку того, что устройство присутствует, что его версия может обрабатываться данным драйвером, что структуры данных драйвера могут быть выделены и инициализированы, и что любое оборудование может быть инициализировано. Драйверы часто сохраняют указатель на свое состояние с помощью *dev_set_drvdata()*. Когда драйвер успешно привязал себя к данному устройству, *probe()* возвращает ноль, и код модели драйвера завершит свою часть привязки драйвера к этому устройству.

Функция *probe()* драйвера может возвращать отрицательное значение *errno*, чтобы указать, что драйвер не связался с этим устройством, и в этом случае он должен был освободить все ресурсы, которые ему были выделены.

Листинг 30: Функция remove

```
int     (*remove)       (struct device * dev);
```

remove() вызывается для отсоединения драйвера от устройства. Она может вызываться, если устройство физически удалено из системы, если модуль драйвера выгружается, во время последовательности перезагрузки или в других случаях.

Драйвер должен определить, присутствует ли устройство или нет. Он должен освободить любые ресурсы, выделенные специально для этого устройства; то есть всё, что было в поле *driver_data* устройства.

Если устройство все еще присутствует, он должно отключить устройство и перевести его в состояние с поддержанием низкого энергопотребления.

Листинг 31: Функция suspend

```
int     (*suspend)      (struct device * dev, pm_message_t state);
```

Suspend() вызывается для перевода устройства в состояние низкого энергопотребления.

Листинг 32: Функция resume

```
int     (*resume)       (struct device * dev);
```

Resume используется для вывода устройства из состояния низкого энергопотребления.

5.9 Атрибуты

Листинг 33: Структура driver_attribute

```
struct driver_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device_driver *driver, char *buf);
        ssize_t (*store)(struct device_driver *, const char * buf, size_t count);
};
```

Драйверы устройств могут экспортировать атрибуты через свои каталоги *sysfs*. Драйверы могут объявлять атрибуты с помощью макроса *DRIVER_ATTR*, который работает идентично макросу *DEVICE_ATTR*.

Пример:

```
DRIVER_ATTR(debug,0644,show_debug,store_debug);
```

Это эквивалентно объявлению:

```
struct driver_attribute driver_attr_debug;
```

Затем это объявление можно использовать для добавления и удаления атрибута из каталога драйвера, используя:

Листинг 34: Файлы атрибутов драйвера

```
int driver_create_file(struct device_driver *, const struct driver_attribute *);
void driver_remove_file(struct device_driver *, const struct driver_attribute *);
```

6 Основы структуры устройства - device.txt

Смотрите *kerneldoc* для получения подробной подробной информации о структуре *device*.

6.1 Интерфейс программирования

Драйвер шины, которая обнаруживает устройство, использует этот вызов для регистрации устройства в ядре:

Листинг 35: Регистрация устройства в драйвере

```
int device_register(struct device * dev);
```

Этот вызов должен инициализировать следующие поля:

- parent
- name
- bus_id
- bus

Устройство удаляется из ядра, когда его счетчик ссылок становится равным 0. Счетчик ссылок можно настроить с помощью:

Листинг 36: Количество ссылок на устройство

```
struct device * get_device(struct device * dev);
void put_device(struct device * dev);
```

get_device() вернет указатель на переданное ему структуру *device*, если ссылка еще не равна 0 (если она уже находится в процессе удаления).

Драйвер может получить доступ к блокировке в структуре устройства, используя:

Листинг 37: Блокировка устройства

```
void lock_device(struct device * dev);
void unlock_device(struct device * dev);
```

6.2 Атрибуты

Листинг 38: Атрибуты устройства

```
struct device_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                        char *buf);
```

```
5         ssize_t (*store)(struct device *dev, struct device_attribute *attr,
6                          const char *buf, size_t count);
7 };
```

Атрибуты устройств могут быть экспортированы драйвером устройства через *sysfs*.

Пожалуйста, смотрите *Documentation/filesystems/sysfs.txt* для получения дополнительной информации о том, как работает *sysfs*.

Как объяснено в *Documentation/kobject.txt*, атрибуты устройства должны быть созданы до того, как будет сгенерировано событие *KOBJ_ADD*. Единственный способ реализовать это - определить группу атрибутов.

Атрибуты объявляются с использованием макроса *DEVICE_ATTR*:

Листинг 39: Макрос атрибута устройства

```
1 #define DEVICE_ATTR(name,mode,show,store)
```

Пример:

Листинг 40: Использование макросов атрибутов устройства

```
1 static DEVICE_ATTR(type, 0444, show_type, NULL);
2 static DEVICE_ATTR(power, 0644, show_power, store_power);
```

Здесь объявляются две структуры типа *struct device_attribute* с соответствующими именами *dev_attr_type* и *dev_attr_power*. Эти два атрибута могут быть организованы в группу следующим образом:

Листинг 41: Группы атрибутов

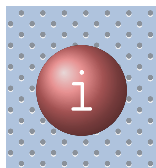
```
1  static struct attribute *dev_attrs[] = {
2          &dev_attr_type.attr,
3          &dev_attr_power.attr,
4          NULL,
5  };
6
7  static struct attribute_group dev_attr_group = {
8          .attrs = dev_attrs,
9  };
10
11 static const struct attribute_group *dev_attr_groups[] = {
12         &dev_attr_group,
13         NULL,
14 };
```

Этот массив групп можно затем связать с устройством, установив указатель группы в *struct device* перед вызовом *device_register()*:

Листинг 42: Назначение группы атрибутов

```
1     dev->groups = dev_attr_groups;
2     device_register(dev);
```

Функция *device_register()* будет использовать указатель *groups* для создания атрибутов устройства, а функция *device_unregister()* будет использовать этот указатель для удаления атрибутов устройства.



Хотя ядро позволяет вызывать *device_create_file()* и *device_remove_file()* в любое время для устройства, пользовательское пространство имеет строгие ограничения в отношении создания атрибутов. Когда новое устройство регистрируется в ядре, генерируется событие для уведомления пользовательского пространства (например, udev) о том, что новое устройство доступно. Если атрибуты добавляются после регистрации устройства, то пользовательское пространство не

будет получать уведомления и пользовательское пространство не будет знать о новых атрибутах.

Это важно для драйвера устройства, который должен создавать дополнительные атрибуты для устройства во время проверки драйвера. Если драйвер устройства просто вызывает *device_ create_file()* для переданной ему структуры устройства, то пользовательское пространство никогда не будет уведомлено о новых атрибутах.

7 Паттерны проектирования драйверов устройств - design-patterns.txt

Этот документ описывает несколько общих шаблонов проектирования, найденных в уже разработанных драйверах устройств. Вполне вероятно, что разработчики подсистем попросят разработчиков драйверов соответствовать этим шаблонам проектирования.

1. Контейнер состояния
2. container_of ()

7.1 State Container

В то время как ядро содержит множество драйверов устройств, которые предполагают, что функция *probe()* будет вызываться только один раз в данной системе (синглтоны), принято считать, что устройство, к которому привязан драйвер, может появится несколько раз. Это означает, что функция *probe()* и все функции обратного вызова должны быть реентерабельными.

Наиболее распространенным способом достижения этого является использование шаблона проектирования контейнера состояния. Этот шаблон обычно имеет такую форму:

Листинг 43: Использование контейнера состояния

```
1  struct foo {
2          spinlock_t lock; /* Example member */
3          (...)
4  };
5
6  static int foo_probe(...)
7  {
8          struct foo *foo;
9
10         foo = devm_kzalloc(dev, sizeof(*foo), GFP_KERNEL);
11         if (!foo)
12         return -ENOMEM;
13         spin_lock_init(&foo->lock);
14         (...)
15 }
```

Этот код будет создавать экземпляр *struct foo* в памяти каждый раз при вызове *probe()*. Это наш контейнер состояния для этого экземпляра драйвера устройства. Конечно, тогда необходимо всегда передавать этот экземпляр состояния всем функциям, которым необходим доступ к состоянию и его членам.

Например, если драйвер регистрирует обработчик прерывания, вы должны передать указатель на *struct foo* следующим образом:

Листинг 44: Передача контейнера сстояний

```
1  static irqreturn_t foo_handler(int irq, void *arg)
2  {
3          struct foo *foo = arg;
4          (...)
5  }
```

```
6
7  static int foo_probe(...)
8  {
9          struct foo *foo;
10
11         (...)
12         ret = request_irq(irq, foo_handler, 0, "foo", foo);
13 }
```

Таким образом, вы всегда получаете указатель на правильный экземпляр *foo* в вашем обработчике прерываний.

7.2 container_of()

Продолжая приведенный выше пример, добавляем работу по разгрузке:

Листинг 45: Уменьшение нагрузки

```
1  struct foo {
2          spinlock_t lock;
3          struct workqueue_struct *wq;
4          struct work_struct offload;
5          (...)
6  };
7
8  static void foo_work(struct work_struct *work)
9  {
10         struct foo *foo = container_of(work, struct foo, offload);
11
12         (...)
13 }
14
15 static irqreturn_t foo_handler(int irq, void *arg)
16 {
17         struct foo *foo = arg;
18
19         queue_work(foo->wq, &foo->offload);
20         (...)
21 }
22
23 static int foo_probe(...)
24 {
25         struct foo *foo;
26
27         foo->wq = create_singlethread_workqueue("foo-wq");
28         INIT_WORK(&foo->offload, foo_work);
29         (...)
30 }
```

Шаблон дизайна одинаков для *hrtimer* или чего-то подобного, который возвращает один аргумент, являющийся указателем на член структуры в функции обратного вызова.

container_of() это просто макрос, определённый в <linux/kernel.h>

Функция *container_of()* предназначена для получении указателя на содержащую структуру из указателя на член этой структуры простым вычитанием с использованием макроса *offsetof()* из стандарта C, который допускает нечто подобное объектно-ориентированному поведению. Обратите внимание, что содержащийся элемент должен быть не указателем, а фактическим элементом, чтобы это работало.

Здесь мы видим, что таким образом мы избегаем глобальных указателей на наш экземпляр *struct foo**, сохраняя при этом количество параметров, передаваемых рабочей функции, в одном указателе.

8 Привязка драйвера - binding.txt

Привязка драйвера - это процесс связывания устройства с драйвером устройства, который может им управлять. Драйверы шины обычно справляются с этим, потому что существуют специфические для шины структуры для представления устройств и драйверов. При использовании общих структур устройств и драйверов устройств большая часть привязки может выполняться с использованием общего кода.

8.1 Шина

Структура типа шины содержит список всех устройств с этим типом шины в системе. Когда *device_register* вызывается для устройства, оно вставляется в конец этого списка. Объект шины также содержит список всех драйверов этого типа шины. Когда *driver_register* вызывается для драйвера, он вставляется в конец этого списка. Это два события, которые запускают привязку драйвера.

8.2 device_register

Когда добавляется новое устройство, перебирается список драйверов шины, чтобы найти то, которое поддерживает это устройство. Чтобы это определить, идентификатор устройства должен соответствовать одному из идентификаторов устройства, поддерживаемых драйвером. Формат и семантика для сравнения идентификаторов зависят от шины. Вместо того, чтобы пытаться получить сложный конечный автомат и алгоритм сопоставления, драйвер шины должен предоставить функцию обратного вызова для сравнения Id устройства с Id драйвера. ШИна возвращает 1, если совпадение найдено; 0 иначе

8.3 int match(struct device * dev, struct device_driver * drv);

Если совпадение найдено, в поле *driver* структуры *device* заносится ссылка на драйвер и делается обратный вызов *probe()* этого драйвера. Этот вызов дает драйверу возможность убедиться, что он действительно поддерживает это аппаратное обеспечение и оно находится в рабочем состоянии.

8.4 Класс устройства

После успешного завершения *probe()* устройство регистрируется в классе, к которому оно принадлежит. Драйверы устройств принадлежат одному и только одному классу, и этот класс задается в поле *devclass* драйвера. *devclass_add_device* вызывается для нумерации устройства в классе и фактической регистрации его в классе, что происходит с помощью функции обратного вызова *register_dev* класса.

8.5 Драйвер

Когда драйвер подключен к устройству, оно вставляется в список устройств драйвера.

8.6 sysfs

В каталоге «устройств» шины создается символическая ссылка, которая указывает на каталог устройства в физической иерархии.

В каталоге драйверов устройства создается символическая ссылка, которая указывает на каталог устройства в физической иерархии.

Каталог для устройства создается в каталоге класса. В этом каталоге создается символическая ссылка, которая указывает на физическое местоположение устройства в дереве *sysfs*.

Символическая ссылка может быть создана (пока этого еще не сделано) в физическом каталоге устройства либо в каталог его класса, либо в каталог верхнего уровня класса. Также можно создать указатель на каталог его драйвера.

8.7 driver_register

Процесс практически идентичен для добавления нового драйвера. Список устройств шины перебирается, чтобы найти соответствие. Устройства, на которых уже есть драйвер, пропускаются. Все устройства перебираются, чтобы связать как можно больше устройств с драйвером.

8.8 Удаление

Когда устройство удаляется, счетчик ссылок для него в конечном итоге становится равным 0. Когда это происходит, вызывается обратный вызов удаления драйвера. Он удаляется из списка устройств драйвера, и счетчик ссылок драйвера уменьшается. Все символические ссылки между ними удалены.

Когда драйвер удаляется, список поддерживаемых им устройств перераспределяется, и для каждого вызывается обратный вызов удаления драйвера. Устройство будет удалено из этого списка, а символические ссылки удалены.

9 Управление ресурсами устройства - devres.txt

6. List of managed interfaces : Currently implemented managed interfaces

9.1 Ведение
A? Devres?

При попытке конвертировать *libata* в *iomap* появился *devres*. Каждый адрес полученный *iomap* должен быть сохранен и освобождён при отключении драйвера. Например, обычный контроллер SFF ATA (то есть старый добрый PCI IDE) в собственном режиме использует 5 PCI BAR, и все они должны поддерживаться.

Как и во многих других драйверах устройств, низкоуровневые драйверы *libata* имеют достаточно много ошибок в *->remove()* и *->probe()* пприводящик к отказам. Что ж, да, возможно, это потому, что разработчики низкоуровневых драйверов libata - ленивые, но не все же разработчики низкоуровневых драйверов?! Проведя день, в возне с аппаратным обеспечением, поврежденным мозгом, без документации или документацией, поврежденной мозгом, если он наконец работает, ну - здорово, это работает!

По той или иной причине драйверы низкого уровня не получают столько внимания и тестирования, как основной код, а ошибки при отключении драйвера или сбое инициализации не случаются достаточно часто, чтобы быть заметными. Путь ведущий к сбою инициализации хуже, потому что он намного короче, в то время как нужно обрабатывать несколько точек входа.

Таким образом, многие низкоуровневые драйверы заканчивают утечкой ресурсов при отсоединении драйвера и имеют наполовину неработающую реализацию пути отказа в *->probe()*, что приводит к утечке ресурсов или даже к возникновению ошибок при сбое. *iomap* добавляет ещё больше к этой мешанине. Так же как и *MSI* и *MSIX*.

9.2 Список Devres
Devres iв двух словах

devres - это в основном связанный список областей памяти произвольного размера, связанных со структурой *device*. Каждая запись *devres* связана с функцией релиза. *devres* может быть создан несколькими способами. Не смотря ни на что, все записи *devres* освобождаются при отключении драйвера. При завершении вызывается связанная с этим функция релиза, а затем запись *devres* освобождается.

Управляемый интерфейс создается для ресурсов, обычно используемых драйверами устройств, использующими *devres*. Например, когерентная память DMA получается с использованием *dma_alloc_coherent()*. Управляемая версия называется *dmam_alloc_coherent()*. Он идентичен *dma_alloc_coherent()*, за исключением того, что память DMA, выделенная с его помощью, управляется и будет автоматически освобождена при отсоединении драйвера. Реализация выглядит следующим образом.

Листинг 46: Реализация управления ресурсами

```
struct dma_devres {
        size_t          size;
        void            *vaddr;
        dma_addr_t      dma_handle;
};

static void dmam_coherent_release(struct device *dev, void *res)
{
        struct dma_devres *this = res;

        dma_free_coherent(dev, this->size, this->vaddr, this->dma_handle);
}

dmam_alloc_coherent(dev, size, dma_handle, gfp)
{
        struct dma_devres *dr;
        void *vaddr;

        dr = devres_alloc(dmam_coherent_release, sizeof(*dr), gfp);
        ...

        /* alloc DMA memory as usual */
        vaddr = dma_alloc_coherent(...);
        ...

        /* record size, vaddr, dma_handle in dr */
        dr->vaddr = vaddr;
        ...

        devres_add(dev, dr);

        return vaddr;
}
```

Если драйвер использует *dmam_alloc_coherent()*, область гарантированно будет освобождена, если инициализация завершится неудачей на полпути или устройство отсоединится. Если большинство ресурсов получено с использованием управляемого интерфейса, драйвер может иметь гораздо более простой код инициализации и выхода. Путь начальной инициализации в основном выглядит следующим образом:

Листинг 47: Путь инициализации

```
my_init_one()
{
        struct mydev *d;

        d = devm_kzalloc(dev, sizeof(*d), GFP_KERNEL);
        if (!d)
        return -ENOMEM;

        d->ring = dmam_alloc_coherent(...);
        if (!d->ring)
        return -ENOMEM;

        if (check something)
        return -EINVAL;
```

```
15          ...
16
17          return register_to_upper_layer(d);
18  }
```

И путь завершения,

Листинг 48: Путь завершения

```
1  my_remove_one()
2  {
3          unregister_from_upper_layer(d);
4          shutdown_my_hardware();
5  }
```

Как показано выше, низкоуровневые драйверы могут быть значительно упрощены с помощью *devres*. Сложность смещена с менее поддерживаемых драйверов низкого уровня на более качественно поддерживаемый верхний уровень. Кроме того, поскольку путь ошибки инициализации используется совместно с путем выхода, оба могут пройти дополнительное тестирование.

9.3 Группы в Devres Сгруппируйте devres-ы и реализуйте их вместе

Записи *Devres* могут быть сгруппированы в группы *devres*. Когда группа освобождается, все содержащиеся в ней записи обычного *devres* и правильно вложенные группы освобождаются. Одним из способов является откат ряда полученных ресурсов при сбое. Например,

Листинг 49: Откат полученных ресурсов

```
1  if (!devres_open_group(dev, NULL, GFP_KERNEL))
2  return -ENOMEM;
3
4  acquire A;
5  if (failed)
6  goto err;
7
8  acquire B;
9  if (failed)
10 goto err;
11 ...
12
13 devres_remove_group(dev, NULL);
14 return 0;
15
16 err:
17 devres_release_group(dev, NULL);
18 return err_code;
```

Поскольку сбой при получении ресурса обычно означает сбой *probe()*, конструкции, подобные приведенным выше, обычно полезны в драйвере среднего уровня (например, уровне ядра *libata*), где функция интерфейса не должна иметь побочного эффекта при сбое. Для LLD в большинстве случаев достаточно просто вернуть код ошибки.

Каждая группа идентифицируется по *void *id*. Он может быть явно указан аргументом *@id* для *devres_ open_ group* или автоматически создан путем передачи значения *NULL* в виде *@id*, как в приведенном выше примере. В обоих случаях *devres_ open_ group()* возвращает идентификатор группы. Возвращенный идентификатор может быть передан другим функциям *devres* для выбора целевой группы. Если для этих функций задано значение *NULL*, выбирается последняя открытая группа.

Например, вы можете сделать что-то вроде следующего:

Листинг 50: Управление группами

```
int my_midlayer_create_something()
{
        if (!devres_open_group(dev, my_midlayer_create_something, GFP_KERNEL))
        return -ENOMEM;

        ...

        devres_close_group(dev, my_midlayer_create_something);
        return 0;
}

void my_midlayer_destroy_something()
{
        devres_release_group(dev, my_midlayer_create_something);
}
```

9.4 Подробности Правила времени жизни, контекст вызова, ...

Время жизни записи *devres* начинается с распределения *devres* и заканчивается, когда она освобождается или уничтожается (удаляется и освобождается) - подсчет ссылок не производится.

Ядро *devres* гарантирует атомарность для всех основных операций *devres* и поддерживает типы *devres* для одного экземпляра (атомарный поиск и добавление, если не найден). Помимо этого, синхронизация одновременного доступа к выделенным данным *devres* является обязанностью вызывающего абонента. Обычно это не проблема, потому что работа шины и распределение ресурсов уже выполняют свою работу.

Для примера *devres* типа одного экземпляра прочитайте *cim_iomap_table()* в *lib/devres.c*.

Все функции интерфейса *devres* могут быть вызваны без контекста, если задана правильная маска gfp.

9.5 Накладные расходы Сколько мы должны заплатить за это?

Каждая учетная информация *devres* распределяется вместе с запрашиваемой областью данных. При отключенной опции отладки учётная информация занимает 16 байтов на 32-битных машинах и 24 байта на 64-битных (три указателя округлены до нулевого выравнивания). Если используется односвязный список, он может быть уменьшен до двух указателей (8 байт на 32 бита, 16 байтов на 64 бита).

Каждая группа *devres* занимает 8 указателей. Её размер может быть уменьшен до 6, если используется односвязный список.

Перерасход памяти на контроллере ahci с двумя портами составляет от 300 до 400 байт на 32-битной машине после нативного преобразования (мы, безусловно, можем потратить немного больше усилий на уровне ядра *libata*).

9.6 Список интерфейсов управления В настоящее время реализованы интерфейсы управления

Листинг 51: Список управляемых интерфейсов

```
MEM
devm_kzalloc()
devm_kfree()

IIO
```

```
6  devm_iio_device_alloc()
7  devm_iio_device_free()
8  devm_iio_trigger_alloc()
9  devm_iio_trigger_free()
10 devm_iio_device_register()
11 devm_iio_device_unregister()
12
13 IO region
14 devm_request_region()
15 devm_request_mem_region()
16 devm_release_region()
17 devm_release_mem_region()
18
19 IRQ
20 devm_request_irq()
21 devm_free_irq()
22
23 DMA
24 dmam_alloc_coherent()
25 dmam_free_coherent()
26 dmam_alloc_noncoherent()
27 dmam_free_noncoherent()
28 dmam_declare_coherent_memory()
29 dmam_pool_create()
30 dmam_pool_destroy()
31
32 PCI
33 pcim_enable_device()      : after success, all PCI ops become managed
34 pcim_pin_device()         : keep PCI device enabled after release
35
36 IOMAP
37 devm_ioport_map()
38 devm_ioport_unmap()
39 devm_ioremap()
40 devm_ioremap_nocache()
41 devm_iounmap()
42 devm_ioremap_resource() : checks resource, requests memory region, ioremaps
43 devm_request_and_ioremap() : obsoleted by devm_ioremap_resource()
44 pcim_iomap()
45 pcim_iounmap()
46 pcim_iomap_table()        : array of mapped addresses indexed by BAR
47 pcim_iomap_regions()      : do request_region() and iomap() on multiple BARs
48
49 REGULATOR
50 devm_regulator_get()
51 devm_regulator_put()
52 devm_regulator_bulk_get()
53 devm_regulator_register()
54
55 CLOCK
56 devm_clk_get()
57 devm_clk_put()
58
59 PINCTRL
60 devm_pinctrl_get()
61 devm_pinctrl_put()
62
63 PWM
64 devm_pwm_get()
65 devm_pwm_put()
66
67 PHY
```

```
68 devm_usb_get_phy()
69 devm_usb_put_phy()
70
71 SLAVE DMA ENGINE
72 devm_acpi_dma_controller_register()
73
74 SPI
75 devm_spi_register_master()
```

10 Портирование существующего драйвера на новую модель - porting.txt

10.1 Обзор

Пожалуйста, обратитесь к документации */driver-model/*.txt* для определения различных типов и концепций драйверов.

Большая часть работы по переносу драйверов устройств на новую модель происходит на уровне драйверов шины. Это было сделано намеренно, чтобы минимизировать негативное влияние на драйверы ядра и обеспечить постепенный переход на драйвера шины.

В двух словах, модель драйвера состоит из набора объектов, которые могут быть встроены в более крупные, специфичные для шины объекты. Поля в этих общих объектах могут заменять поля в объектах, связанных с шиной.

Общие объекты должны быть зарегистрированы в модели драйвера ядра. Таким образом они будут экспортированы через файловую систему *sysfs*. *sysfs* можно подключить, выполнив

```
1 # mount -t sysfs sysfs /sys
```

10.2 Процесс портирования

10.2.1 Step 0: Подготовка

Внимательно изучите *include/linux/device.h* на предмет определений объектов и функций.

10.2.2 Step 1: Регистрация драйвера шины.

Листинг 52: Структура bus_type

Определяем структуру *bus_type* для драйвера шины.

```
1 struct bus_type pci_bus_type = {
2         .name           = "pci",
3 };
```

Регистрируем тип шины Это должно быть сделано в функции инициализации для типа шины, которая обычно является функцией *module_init()*, или эквивалентной функцией.

Листинг 53: Регистрация типа шины

```
1 static int __init pci_driver_init(void)
2 {
3         return bus_register(&pci_bus_type);
4 }
5
6 subsys_initcall(pci_driver_init);
```

Тип шины можно разрегистрировать (если драйвер шины может быть скомпилирован как модуль), выполнив:

Листинг 54: Разрегистрация типа шины

```
1 bus_unregister(&pci_bus_type);
```

Экспортируем тип шины для последующего использования Другой код может ссылаться на этот тип шины, поэтому объявите его в общем файле заголовка и экспортируйте этот символ.

Из *include/linux/pci.h*:

Листинг 55: Объявление типа шины

```
1 extern struct bus_type pci_bus_type;
```

Указанный выше код из файла можно экспоритровать так:

Листинг 56: Экспорт типа шины

```
1 EXPORT_SYMBOL(pci_bus_type);
```

Почему в каталоге шины /sys/bus/pci/ with два подкаталога 'devices' и 'drivers'.

```
# tree -d /sys/bus/pci/
/sys/bus/pci/
|-- devices
`-- drivers
```

10.2.3 Step 2: Регистрация устройства.

Структура *device* представляет одно устройство. В основном она содержит метаданные, описывающие связь устройства с другими объектами.

Встраивание структуры *device* в устройство, специфичное для шины

```
1 struct pci_dev {
2         ...
3         struct  device  dev;            /* Generic device interface */
4         ...
5 };
```

Рекомендуется, чтобы универсальная структура *device* не было первым элементом в специфичной структуре, чтобы отговорить программистов от бессмысленного приведения типов объектов. Вместо этого должны быть созданы макросы или встроенные функции для преобразования из универсального типа объекта.

Листинг 57: Макрос преобразования типов

```
1 #define to_pci_dev(n) container_of(n, struct pci_dev, dev)
```

или

Листинг 58: Функция преобразования типов

```
1 static inline struct pci_dev * to_pci_dev(struct kobject * kobj)
2 {
3         return container_of(n, struct pci_dev, dev);
4 }
```

Это позволяет компилятору проверять безопасность типов выполняемых операций (что хорошо).

Инициализация устройства во время регистрации Когда устройства обнаруживаются или регистрируются задавая тип шины, драйвер шины должен инициализировать общее устройство. Наиболее важными для инициализации являются поля *bus_id*, *parent* и *bus*.

Bus_id - это строка ASCII, которая содержит адрес устройства на шине. Формат этой строки зависит от шины. Она необходима для представления устройств в *sysfs*.

parent является физическим родителем устройства. Важно, чтобы драйвер шины устанавливал это поле правильно.

Модель драйвера поддерживает упорядоченный список устройств, который она использует для управления питанием. Этот список нужен для того, чтобы гарантировать, что устройства отключаются раньше, чем их физические родители, и наоборот. Порядок этого списка определяется родителем зарегистрированных устройств.

Кроме того, расположение устройства в каталоге *sysfs* зависит от родителя устройства. *sysfs* экспортирует структуру каталогов, отражающую иерархию устройств. Точная установка родительского элемента гарантирует, что *sysfs* будет точно представлять иерархию.

Поле шины устройства является указателем на тип шины, к которой принадлежит устройство. Оно должно быть установлено значением *bus_type*, которое было объявлено и инициализирован ранее.

При желании драйвер шины может установить поля имени устройства и его версии.

Поле имени представляет собой строку ASCII, описывающую устройство, например

"ATI Technologies Inc Radeon QD"

Поле *release* является обратным вызовом, который ядро модели драйвера вызывает, когда устройство удаляется, и все ссылки на него были освобождены. Подробнее об этом чуть ниже.

Регистрация устройства После инициализации общего устройства его можно зарегистрировать в модели драйвера ядра, выполнив:

```
1 device_register(&dev->dev);
```

Позднее его можно разрегистрировать, вызвав:

```
1 device_unregister(&dev->dev);
```

Такое должно происходить на шинах, которые поддерживают устройства с возможностью горячего подключения. Если драйвер шины отменяет регистрацию устройства, ядро не должно немедленно его освобождать. Вместо этого следует дождаться, пока модель драйверов ядра вызовет метод освобождения устройства, а затем освободить объект, относящийся к шине. (Может быть имеется другой код, который в настоящее время ссылается на структуру устройства, и было бы грубой ошибкой освобождать устройство, пока происходит нечто подобное).

Когда устройство зарегистрировано, каталог в *sysfs* создается. Дерево PCI в *sysfs* выглядит так:

```
/sys/devices/pci0/
|-- 00:00.0
|-- 00:01.0
|   `-- 01:00.0
|-- 00:02.0
|   `-- 02:1f.0
|       `-- 03:00.0
|-- 00:1e.0
|   `-- 04:04.0
|-- 00:1f.0
|-- 00:1f.1
|   |-- ide0
|   |   |-- 0.0
|   |   `-- 0.1
```

```
|   `-- ide1
|       `-- 1.0
|-- 00:1f.2
|-- 00:1f.3
`-- 00:1f.5
```

Кроме того, символические ссылки создаются в каталоге *devices* шины, они указывают на каталог устройства в физической иерархии.

```
/sys/bus/pci/devices/
|-- 00:00.0 -> ../../../devices/pci0/00:00.0
|-- 00:01.0 -> ../../../devices/pci0/00:01.0
|-- 00:02.0 -> ../../../devices/pci0/00:02.0
|-- 00:1e.0 -> ../../../devices/pci0/00:1e.0
|-- 00:1f.0 -> ../../../devices/pci0/00:1f.0
|-- 00:1f.1 -> ../../../devices/pci0/00:1f.1
|-- 00:1f.2 -> ../../../devices/pci0/00:1f.2
|-- 00:1f.3 -> ../../../devices/pci0/00:1f.3
|-- 00:1f.5 -> ../../../devices/pci0/00:1f.5
|-- 01:00.0 -> ../../../devices/pci0/00:01.0/01:00.0
|-- 02:1f.0 -> ../../../devices/pci0/00:02.0/02:1f.0
|-- 03:00.0 -> ../../../devices/pci0/00:02.0/02:1f.0/03:00.0
`-- 04:04.0 -> ../../../devices/pci0/00:1e.0/04:04.0
```

10.2.4 Step 3: Registering Drivers.

struct device_driver - это простая структура драйвера, которая содержит набор операций, которые ядро может вызывать через модель драйвера.

Встраивание структуры *device_driver* в драйвер, специфичный для шины. Так же, как с устройствами, надо сделать что-то вроде:

```
1 struct pci_driver {
2         ...
3         struct device_driver     driver;
4 };
```

Инициализация общей структуры драйвера Когда драйвер регистрируется на шине (например, выполняет *ci_register_driver()*), инициализируйте необходимые поля структуры *driver*: *name* и *bus*.

Регистрация драйвера. После того, как универсальный драйвер был инициализирован, вызовите

```
1 driver_register(&drv->driver);
```

чтобы зарегистрировать драйвер в ядре.

Если драйвер не зарегистрирован на шине, отмените его регистрацию в ядре, выполнив:

```
1 driver_unregister(&drv->driver);
```

Обратите внимание, что этот вызов будет блокировать поток управления, пока не исчезнут все все ссылки на этот драйвер. Обычно их не бывеает.

Представление в Sysfs Драйверы экспортируются через *sysfs* в каталоге драйверов их шин. Например:

```
/sys/bus/pci/drivers/
|-- 3c59x
|-- Ensoniq AudioPCI
|-- agpgart-amdk7
|-- e100
`-- serial
```

10.2.5 Step 4: Define Generic Methods for Drivers.

struct device_ driver определяет набор операций, которые вызывает модель драйверов ядра. Большинство из этих операций, вероятно, аналогичны операциям, которые шина уже определяет для драйверов, но принимают другие параметры.

Было бы сложно и утомительно заставить каждый драйвер на шине одновременно конвертировать их драйверы в общий формат. Вместо этого драйвер шины должен определять единичные экземпляры универсальных методов, которые перенаправляют вызов на драйверы, специфичные для шины. Например:

```
1  static int pci_device_remove(struct device * dev)
2  {
3          struct pci_dev * pci_dev = to_pci_dev(dev);
4          struct pci_driver * drv = pci_dev->driver;
5
6          if (drv) {
7                  if (drv->remove)
8                          drv->remove(pci_dev);
9                  pci_dev->driver = NULL;
10         }
11         return 0;
12 }
```

Универсальный драйвер должен быть инициализирован этими методами до его регистрации.

```
1          /* initialize common driver fields */
2          drv->driver.name = drv->name;
3          drv->driver.bus = &pci_bus_type;
4          drv->driver.probe = pci_device_probe;
5          drv->driver.resume = pci_device_resume;
6          drv->driver.suspend = pci_device_suspend;
7          drv->driver.remove = pci_device_remove;
8
9          /* register with core */
10         driver_register(&drv->driver);
```

В идеале шина должна инициализировать поля, только если они еще не установлены. Это позволяет драйверам реализовывать свои собственные общие методы.

10.2.6 Step 5: Support generic driver binding.

Модель предполагает, что устройство или драйвер могут быть динамически зарегистрированы на шине в любое время. Когда происходит регистрация, устройства должны быть связаны с драйвером, или драйверы должны быть связаны со всеми устройствами, которые он поддерживает.

Драйвер обычно содержит список идентификаторов устройств, которые он поддерживает. Драйвер шины сравнивает эти идентификаторы с идентификаторами устройств, зарегистрированных с ним. Формат идентификаторов устройств и семантика их сравнения зависят от шины, поэтому общая модель пытается их обобщить.

Вместо этого шина может предоставить метод в структуре *bus_ type*, который выполняет сравнение:

```
1 int (*match)(struct device * dev, struct device_driver * drv);
```

Функция *match* должна вернуть «1», если драйвер поддерживает устройство, и «0» в противном случае.

Когда устройство зарегистрировано, перебирается список драйверов шины. *bus-> match()* вызывается для каждого драйвера, пока не будет найдено совпадение.

Когда драйвер зарегистрирован, перебирается список устройств шины. *bus-> match()* вызывается для каждого устройства, которое еще не захвачено драйвером.

Когда устройство успешно связано с драйвером, *device->driver* установлено, устройство добавляется в список устройств для каждого драйвера, и в каталоге драйвера *sysfs* создается символическая ссылка, которая указывает на физический каталог устройства:

```
/sys/bus/pci/drivers/
|-- 3c59x
|   ‘-- 00:0b.0 -> ../../../../devices/pci0/00:0b.0
|-- Ensoniq AudioPCI
|-- agpgart-amdk7
|   ‘-- 00:00.0 -> ../../../../devices/pci0/00:00.0
|-- e100
|   ‘-- 00:0c.0 -> ../../../../devices/pci0/00:0c.0
‘-- serial
```

Эта привязка драйвера должна заменить существующий механизм привязки драйвера, используемый в данный момент на шине.

10.2.7 Step 6: Supply a hotplug callback.

Всякий раз, когда устройство регистрируется в модели драйвера ядра, вызывается программа из пользовательского пространства */sbin/hotplug* для уведомления пользователя. Пользователи могут определять действия, выполняемые при вставке или удалении устройства.

Модель драйвера ядра передает несколько аргументов в пространство пользователя через переменные среды, включая

- *ACTION*: задаёт 'add' или 'remove'
- *DEVPATH*: задаёт физический путь к устройству в *sysfs*.

Драйвер шины также может предоставлять дополнительные параметры для использования пользователем. Для этого шина должна реализовать метод *hotplug* в *struct bus_ type*:

```
1 int (*hotplug) (struct device *dev, char **envp,
2                 int num_envp, char *buffer, int buffer_size);
```

Который вызывается непосредственно перед выполнением */sbin/hotplug*.

10.2.8 Step 7: Cleaning up the bus driver.

Общие структуры шины, устройства и драйвера предоставляют несколько полей, которые могут заменить те, которые определены для драйвера шины приватно.

Список устройств. *struct bus_ type* содержит список всех устройств, зарегистрированных с данным типом шины. Она включает в себя все устройства на всех экземплярах этого типа шины. Внутренний список, который использует сама шина, может быть удален в пользу использования этого списка.

Ядро предоставляет итератор для доступа к этим устройствам:

```
1 int bus_for_each_dev(struct bus_type * bus, struct device * start,
2                      void * data, int (*fn)(struct device *, void *));
```

Список драйверов. *struct bus_ type* также содержит и список всех драйверов, зарегистрированных на этой шине. Можно отказаться от использования внутреннего списка драйверов, поддерживаемого драйвером шины, в пользу использования общего.

Драйверы можно перебирть точно так же, как устройства:

```
1 int bus_for_each_drv(struct bus_type * bus, struct device_driver * start,
2                      void * data, int (*fn)(struct device_driver *, void *));
```

Пожалуйста, посмотрите *drivers/base/bus.c* для получения более полной информации.

rwsem *struct bus_ type* содержит семафор *rwsem*, который защищает доступ к спискам устройств и драйверов. Он может использоваться внутренним драйвером шины и должен использоваться при доступе к списку устройств или драйверов, поддерживаемых шиной.

Поля в *device* и *driver*. Некоторые поля в *struct device* и *struct device_ driver* дублируют поля в шинных представлениях этих объектов. Не стесняйтесь удалять специфичные для шины и предпочитайте общие. Обратите внимание, что это, вероятно, будет означать исправление всех драйверов, которые ссылаются на специфичные для шины поля (хотя все они должны быть ов одну строку).

Это приложение к основному документу