

OpenPOD: Открытая спецификация API переносимых драйверов

Версия API - 0.1

Версия спецификации - 0.1

Комментарии присылать Дмитрию Завалишину на dz@dz.ru

Данная спецификация разрабатывается сообществом Российских разработчиков операционных систем во исполнение договорённостей, достигнутых на конференции OS Day Иннополис 2015. Спецификация открыта для комментариев. Настоящая версия является первым драфтом, и будет разрабатываться на русском языке до достижения консенсуса по основному содержанию. Далее планируется перейти на английский язык как основной язык документа (русская версия будет поддерживаться как перевод с английского).

Disclaimer от dz: дамы и господа, я “вылил” этот документ из головы, прошу не судить строго, но отнестись конструктивно. Это - самый первый черновик, который я предлагаю как заправку для начала работы. Документ явно будет реструктуризирован - разбит на overview и детализацию + примеры кода. Сейчас предлагаю сосредоточиться на сути. Я прислал вам ссылку, которая позволяет не править текст, но оставлять комментарии к нему. Для размещения комментария необходимо выделить спорный фрагмент текста, нажать правую кнопку и выбрать пункт меню “комментарий”, а в комментарии изложить сомнения или свой вариант. Чужие комментарии можно комментировать.

Цели

Спецификация призвана обеспечить возможность написания драйверов, переносимых между различными операционными системами.

Ограничения

В настоящий момент спецификация верифицируется только для графических драйверов (frame buffer). Следующим этапом предполагается описание драйверов блочных, сетевых и символьных устройств.

Данная спецификация описывает собственно API (и, возможно, ABI) драйвера, плюс описывает и содержит референсную реализацию хелпер-функций и компонент для поддержки части описанных интерфейсов. Референсная реализация не входит в спецификацию как таковую, и необязательна.

Общее описание

Спецификация описывает программный интерфейс (API). В последующих версиях планируется описание бинарного интерфейса (ABI). Реализация ABI опциональна.

Данный документ представляет собой обзор спецификации. Параллельно будут представлены подробный референс по всем функциям, хедер-файлы и референсная реализация функций фреймворка.

Документ специфицирует только интерфейс - функции фреймворка предоставляются as is, без гарантии и не являются единственной возможной реализацией. Разработчик ядра волен использовать их, или переписать их самостоятельно.

Термины

Драйвер - программная компонента, отвечающая за взаимодействие с определённым аппаратным обеспечением компьютера. Никакие два драйвера не должны работать с одним и тем же подмножеством аппаратуры одновременно.

Устройство - интерфейс драйвера к определённому функциональному подмножеству обслуживаемой драйвером аппаратуры. Например, драйвер дисковой подсистемы может экспортировать в ядро несколько устройств

Лицензия

TBD

Референсные хедер-файлы и код, прилагаемые к данной спецификации, распространяются по лицензии LGPL.

Авторы и контрибуторы

Основная рабочая группа:

Дмитрий Завалишин, dz@dz.ru - оригинальный драфт
Алексей Хорошилов, khoroshilov@ispras.ru
Кан Анна, kan_a@mail.ru
Мезенцев Илья, i.mezentsev@wisetech.pro
Годунов Александр, nkag@niisi.ras.ru

Антон Бондарев, anton.bondarev2310@gmail.com

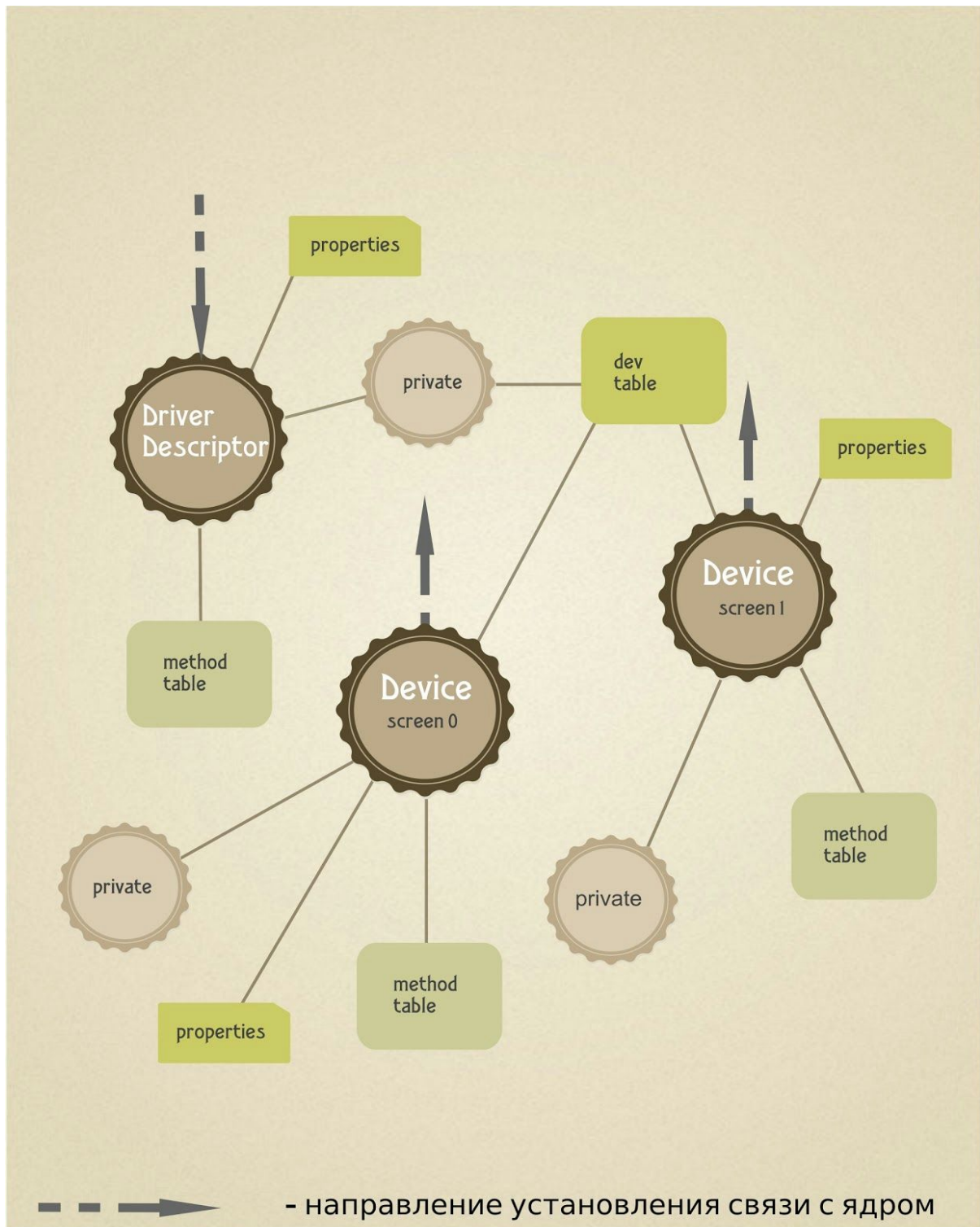
Петренко Александр, a.k.petrenko@gmail.com

Контрибуторы

Здесь Могла Бы Быть Ваша Фамилия :)

Общая структура драйвера

Драйвер экспортирует в ядро интерфейс управления драйвером в целом и одно или несколько устройств, каждое из которых имеет определённый класс и, соответственно, экспортирует как универсальный, так и специфичный для класса интерфейс. Драйвер может быть динамически загружен, запущен, остановлен и выгружен (предусмотреть признак невыгружаемости драйвера?). Устройства динамически синтезируются драйвером в результате сканирования фактической аппаратуры, и могут появляться и исчезать во время работы. Драйвер может экспортировать в ядро устройства разных классов (например, драйвер IBM PS/2 keyboard and mouse будет экспортировать два устройства различных классов).



К спецификации прилагаются примеры драйверов и два скелета - для разработки минимального драйвера (одно статическое устройство без обнаружения, очереди запросов и нитей, несколько динамических устройств с поиском по PCI, нитями и очередью запросов).

Классы драйверов и устройств

Драйвер в целом и экспортируемые драйвером интерфейсы конкретных устройств должны быть отнесены к определённом классу, идентифицирующему специфический для данного типа устройств интерфейс ядра.

Если все устройства, которые экспортирует данный драйвер, единообразны, должен быть указан конкретный класс драйвера. Иначе для драйвера указывается класс **POD_DEV_CLASS_MULTIPLE**, а для устройства - конкретный класс.

```
#define POD_DEV_CLASS_VOID          0 // undefined or none
#define POD_DEV_CLASS_SPECIAL       1 // has user-defined non-standard interface
#define POD_DEV_CLASS_VIDEO         2 // framebuf, bitblt io
#define POD_DEV_CLASS_BLOCK         3 // disk, cdrom - block io
#define POD_DEV_CLASS_CHARACTER     4 // tty, serial, byte io
#define POD_DEV_CLASS_NET           5 // packet IO, has MAC address

#define POD_DEV_CLASS_KEYBD         6 // key (make/break) events
#define POD_DEV_CLASS_MOUSE         7 // mouse coordinate events

#define POD_DEV_CLASS_MULTIPLE      0xFF // driver only, has multiple dev types
```

Основной интерфейс драйвера

Lifecycle интерфейс

Определяет процесс инициализации, активации, сканирования аппаратуры, инициализации точек связи с ядром, деактивации и т.п.

pod_construct - инициализация структур данных драйвера, “статическая” связка с ядром ОС (подключение API, etc), но не сканирование или инициализация аппаратуры.

```
errno_t      pod_construct( pod_driver *drv ); // ENOMEM
```

pod_destruct - полное завершение работы драйвера (далее может следовать выгрузка). Должны быть закончены все нити, возвращены все ресурсы. Предполагается, что перед вызовом этого метода вызван deactivate. В противном случае destruct выполняет минимально возможные и наиболее жёсткие действия, аналогичные вызову deactivate.

```
errno_t      pod_destruct( pod_driver *drv ); // err?
```

pod_activate - инициализировать аппаратуру, запустить драйвер. До этого должно пройти сканирование аппаратуры. До запуска activate драйвер возвращает ошибку на

любой операционный вызов (запрос на ввод-вывод) и не инициирует обмен с ядром со своей стороны.

errno_t pod_activate(pod_driver *drv); // ENOMEM

pod_deactivate - остановить драйвер, деинициализировать аппаратуру.

errno_t pod_deactivate(pod_driver *drv); // err?

pod_sense - ядро просит драйвер произвести поиск устройств, но не инициализировать их. В рамках этого вызова драйвер может произвести цикл запросов к ядру для сканирования аппаратуры.

errno_t pod_sense(pod_driver *drv); // err?

pod_offer - ядро предлагает драйверу (PCI?) устройство, которое, по мнению ядра, этому драйверу соответствует. Драйвер верифицирует устройство и принимает или отказывается от обслуживания.

errno_t pod_offer(pod_driver *drv, bus?, dev?); // ENOMEM, EFAULT

Основной сценарий:

Ядро вызывает **pod_construct**, драйвер инициализирует структуры данных.

Ядро вызывает **pod_sense** и/или **pod_offer**, драйвер обследует аппаратуру и определяется с составом обслуживаемых устройств.

Ядро вызывает **pod_activate**, драйвер регистрирует в ядре известные ему устройства (**pod_dev_link**) и активируется.

Нормальная работа драйвера. Если в процессе работы драйвер обнаруживает или теряет устройства, происходят вызовы **pod_dev_link/pod_dev_unlink**.

Ядро вызывает **pod_deactivate**, драйвер останавливает (и deregистрирует?) все устройства.

Ядро вызывает **pod_destruct**, драйвер освобождает все занятые ресурсы и может быть выгружен.

Работа драйвера:

Для каждого известного ему устройства драйвер вызывает **pod_ev_link**.

Нормальная работа устройства.

Драйвер вызывает **pod_dev_unlink**.

Работа устройства, в основном, состоит из вызовов **rq_start** ядром и ответных коллбеков драйвера по исполнению запроса. Если устройство предоставляет специфический для класса набор методов (поле **class_interface**), ядро может, также, вызывать эти методы. Пример специфического метода - **get_mac_address** для драйвера сетевого устройства.

Иллюстрации для стейт-машины TBD

Операционный интерфейс

Все точки входа этого раздела должны вызываться в контексте процесса/нити (не прерывания), могут быть кратковременно заблокированы (для работы с очередями), и не могут быть долговременно заблокированы. Первым параметром каждого вызова является структура устройства:

```
struct pod_device {
    char                magic[4];

    uint8_t             class;
    uint8_t             pad0;
    uint8_t             pad1;
    uint8_t             pad2;

    pod_driver          *drv;

    dev_f               *calls; // dev io entry points

    pod_properties      *prop;

    void                *class_interface; // dev class specific interface
    void                *private;

    // -----
    // Fields below are used by default framework code

    // Request queue, used by pod_dev_q_ functions
    pod_q               *default_r_q;      // default request q
    pod_request         *curr_rq;         // request we do now
    pod_thread          *rq_run_thread;    // thread used to run requests
    pod_cond            *rq_run_cond;      // triggered to run next request
};
```

pod_dev_stop - остановить устройство (например, чтобы отсоединить или выключить)

pod_dev_start - запустить устройство

errno_t **pod_dev_stop(pod_device *dev);**

errno_t **pod_dev_start(pod_device *dev);**

pod_rq_start - запросить ввод-вывод

pod_rq_dequeue - снять запрос, если это ещё возможно (фактически, вызов необязательный - имеет право всегда говорить, что уже поздно)

pod_rq_fence - гарантировать, что все предыдущие запросы будут отработаны до любого последующего

pod_rq_raise - изменить (обычно - повысить:) приоритет запроса. Необязательно к реализации, но отсутствие может серьёзно влиять на отзывчивость системы.

done - внутри структуры запроса ввода-вывода есть указатель на функцию, которая будет вызвана по исполнению запроса. Функция **done** может быть вызвана контексте прерывания. Ядро должно самостоятельно обеспечить проверку на контекст прерывания и, при необходимости, конвертацию вызова в контекст нити ядра или иной уместный контекст.

```
struct pod_request {
    uint8_t      request_class;    // block/character/net/video io
    uint8_t      operation;        // op id - read/write/clear screen/etc

    // request priority, more = earlier
    uint32_t     io_prio;

    pod_rq_status err;             // результат исполнения, 0 = ок

    // Req struct part specific for request class and op id
    void          *class_specific;

    // Driver will call this when done
    void          (*done)( pod_request *rq);
};
```

// ENOMEM если очередь переполнена

errno_t **pod_rq_start(pod_device *dev, pod_request *rq);**

// если ещё не сделано - остановить

errno_t **pod_rq_dequeue(pod_device *dev, pod_request *rq);**

// гарантировать, что предыдущие запросы будут отработаны до исполнения
следующих

errno_t **pod_rq_fence(pod_device *dev, pod_request *rq);**

// изменить приоритет

errno_t **pod_rq_raise(pod_device *dev, pod_request *rq, uint32_t io_prio);**

Комментарий: в данной спецификации предполагается, что драйвер самостоятельно хранит и обрабатывает (сортирует и т.п.) очередь запросов. Решение принято в связи с тем, что современная аппаратура зачастую готова реализовывать очереди аппаратно. Кроме того, это необходимо для драйверов virtio.

Стандартные хелперы (обеспечиваются фреймворком)

Очередь запросов, базовая реализация.

// создать очередь

errno_t **pod_q_construct(pod_q **q);**

// уничтожить очередь

errno_t **pod_q_destruct(pod_q *q);**

// поставить запрос в очередь

errno_t **pod_q_enqueue(pod_q *q, pod_request *rq);**

// забрать запрос из очереди

errno_t **pod_q_dequeue(pod_q *q, pod_request *rq);**

// выставить ограду

errno_t **pod_q_fence(pod_q *q, pod_request *rq);**

// отсортировать с учётом оград

errno_t **pod_q_sort(pod_q *q, pod_request *rq, int (*cmp)(*rqa, *rqb));**

// функция сравнения по приоритету

int **rq_prio_cmp(pod_request *rqa, pod_request *rqb)**

Комплект функций фреймворка, которые реализуют дефолтный механизм работы с очередью запросов - для типового драйвера с одной очередью.

```

errno_t      pod_dev_q_construct( pod_device *dev );
errno_t      pod_dev_q_destruct( pod_device *dev );
errno_t      pod_dev_q_enqueue( pod_device *dev, pod_request *rq );
errno_t      pod_dev_q_dequeue( pod_device *dev, pod_request *rq );
errno_t      pod_dev_q_fence( pod_device *dev, pod_request *rq );
errno_t      pod_dev_q_raise( pod_device *dev, pod_request *rq, uint32_t io_prio );

```

// должна быть реализована в драйвере, запускает запрос на исполнение

```

errno_t      pod_dev_q_exec( pod_device *dev, pod_request *rq );

```

// реализована в фреймворке, вызывается кодом драйвера по завершении в/в, можно из прерывания

```

errno_t      pod_dev_q_iodone( pod_device *dev );

```

Запросы для видеодрайверов

Запросы отрабатываются через pod_request (структура передаётся через указатель class_specific), или через таблицу методов class_interface устройства.

// значение поля operation или индекс в таблицу методов class_interface

```

enum pod_video_operations {
    nop, getmode, setmode,
    clear_all, clear,
    move,
    write, read,
    write_part, read_part
};

```

// формат пикселя

```

enum pod_pixel_fmt {
    pod_pixel_rgb,      // 24 бита
    pod_pixel_rgba,     // 32 бита, a is ignored by hw
    pod_pixel_r5g6b5,   // 16 бит, 5-6-5
    pod_pixel_r5g5b5,   // 16 бит, 5-5-5
};

```

// флаги блиттера (режим копирования пикселей на экран - с экрана читаем всегда!)

```

enum pod_v_flags {
    pod_video_ignore_zbuffer, // игнорировать z координату
    pod_video_ignore_alpha,  // игнорировать альфа-канал
};

```

// clear

```

struct pod_video_rq_square

```

```

{
    uint32_t    x, y;
    uint32_t    x_size, y_size;
};

```

// move from screen to screen

```

struct pod_video_rq_2square
{
    uint32_t    from_x, from_y;
    uint32_t    from_x_size, from_y_size;

    uint32_t    to_x, to_y;
    uint32_t    to_x_size, to_y_size;
};

```

// write, read - пересылка полного битмапа на экран или с экрана

```

struct pod_video_rq_rw
{
    uint32_t    x, y;
    uint32_t    x_size, y_size;

    uint32_t    z;
    pod_v_flags flags;

    char        *buf;

    pod_pixel_fmt    buf_fmt;    // формат пикселя буфера ядра
};

```

// write_part, read_part - пересылка части битмапа на экран или с экрана

```

struct pod_video_rq_rw_part
{
    uint32_t    from_x, from_y;                // point to start in buf
    uint32_t    from_x_size, from_y_size;    // full size of bitmap in buf

    uint32_t    to_x, to_y;                    // point to start on screen

    uint32_t    move_x_size, move_y_size;    // size of sqare to move

    uint32_t    z;                            // z position
    pod_v_flags flags;

    char        *buf;

    pod_pixel_fmt    buf_fmt;    // формат пикселя буфера ядра
};

```

```

// getmode, setmode
struct pod_video_rq_mode
{
    // если невалидно - ставим ближайший вверх, если нет - ошибка
    uint32_t      x_size, y_size;
    // формат пикселя экрана, желательно rgba
    pod_pixel_fmt      buf_fmt;

    // Возвращаемое значение.
    physaddr_t  vbuf;          // может быть 0? тогда всё через методы?
};

```

Запросы для дисковых драйверов

```

// значение поля operation или индекс в таблицу методов class_interface
enum pod_block_operations {
    nop, read, write, trim
};

// trim
struct pod_block_rq
{
    diskaddr_t  block_no;
    uint32_t    block_count;
    uint32_t    block_size;
};

// read, write
struct pod_block_io_rq
{
    diskaddr_t  block_no;
    uint32_t    block_count;
    uint32_t    block_size;
    physaddr_t  physmem_addr;
};

```

Дополнительный интерфейс драйвера

Не обязателен к реализации. Точки входа должны присутствовать, но могут работать частично или не работать вообще. Неработающие функции или подфункции должны возвращать разумный код ошибки.

Свойства (rich man's ioctl)

Именованные свойства драйвера или устройства.

```
errno_t      (*pod_getproperty)( pod_prop *p, const char *pName, char *pValue, int
vlen );
errno_t      (*pod_setproperty)( pod_prop *p, const char *pName, const char
*pValue );
errno_t      (*pod_listproperties)( pod_prop *p, int nProperty, char *pName, int vlen
);
```

pod_getproperty - получить значение свойства. Вызов не может приводить к изменению состояния драйвера или устройства.

pod_setproperty - установить значение свойства. Может иметь сайд-эффект по изменению состояния драйвера или устройства (фактически - может являться вызовом метода без параметров.)

pod_listproperties - Получить имя очередного свойства. Для получения всех имён свойств вызывается циклически, до получения ненулевого (ошибочного) значения функции.

pName - имя свойства, ascii строка, завершённая нулём.
pValue - значение свойства, ascii строка, завершённая нулём.
vlen - размер буфера для получения значения или имени.

Возвращаемые значения

ENOENT - нет свойства с указанным именем или (для listproperties) нет больше свойств.

EFAULT - не удалось получить значение свойства (но имя свойства верное)

EINVAL - не удалось установить значение свойства (но имя свойства верное), недопустимое значение свойства.

Сохранение и восстановление состояния драйвера

pod_save_state - сохранить состояние драйвера в XML. Выполняется deactivate, но полное состояние драйвера и устройства сохраняется.

pod_load_state - состояние восстанавливается из XML, выполняется activate.

Между вызовами возможна полная остановка аппаратуры и перезагрузка ОС.

Управление энергопотреблением

pod_sleep - аппаратура устройства переводится в режим экономии энергии. драйвер должен быть деактивирован. Возможно указание степени засыпания - в частности, в терминах скорости восстановления.

pod_awake - аппаратура переводится в нормальный рабочий режим.

Интерфейс ядра ОС

Ядро ОС должно предоставлять драйверу описанный ниже набор функций. Драйвер не должен использовать никакие иные точки входа в ядро.

При статической линковке или если технология загрузки загружаемого драйвера допускает динамическую привязку к экспортируемым функциям ядра, доступ к точкам входа ядра осуществляется напрямую. Если это невозможно, ядро должно прописать в поле **kernel_driver_api** таблицу указателей на соответствующие функции.

TBD

Специфический интерфейс ядра для драйвера

pod_dev_link - драйвер сообщает ядру о наличии/появлении устройства, ядро подключает устройство к себе и, если нужно, запускает его

pod_dev_unlink - драйвер сообщает, что устройство более не валидно. После возврата из этого вызова драйвер имеет право деаллоцировать структуры данных устройства, видимые ядру.

pod_dev_event - репорт о событии (ошибке, отказе?) устройства

Управление ресурсами

pod_get_bus_list - получить все имеющиеся в железе шины
pod_get_pci_bus - возвращает главную шину PCI
pod_get_usb_bus - ... USB
...more...

pod_bus_scan - искать устройство на шине по определённым параметрам
pod_bus_enum - перечислить все устройства на шине

pod_bus_reserve_resource - захватить ресурс (порт, память, прерывание)
pod_bus_release_resource - освободить ресурс

Управление памятью

pod_alloc_physmem - выделяет физическую память без отображения, постранично
pod_alloc_kheap - выделяет виртуальную память ядра (не пейджируемую)
pod_alloc_vaddr - выделяет виртуальное адресное пространство без памяти

pod_free_physmem
pod_free_kheap
pod_free_vaddr

map/unmap physmem (kernel can verify that driver maps just what it allocated)

pod_map_mem - физ адрес, вирт адрес, флаг кешируемости?
pod_unmap_mem

// нужно это? или требовать от ядра, чтобы все запросы приходили на привязанную память? для блочных оно так и есть - в запросе вообще приходит физпамять.

pod_wire_mem - для неvirtуальной памяти пор, виртуальную пейджирует в, гарантирует от вытеснения (память превращается в непейджируемую) - НЕЛЬЗЯ вызывать в контексте точки входа, только из треда драйвера.
pod_unwire_mem - отменить непейджируемость

Логирование

syslog?

Нити и синхронизация

В целом планируется в рамках pthread, дополнения приветствуются.

Загрузка загружаемого или привязка статического драйвера

Структура дескриптора драйвера

```
pod_driver {
    char                magic[4];    // magic number

    // Minor change means API is extended, but compatible
    uint8_t             API_version_major;
    uint8_t             API_version_minor;

    // i32, i64, arm32, arm64, mips32, mips64, ...
    uint8_t             arch_major;
    uint8_t             arch_minor; // undefined, must be 0

    uint8_t             class;
    uint8_t             pad0;
    uint8_t             pad1;
    uint8_t             pad2;

    //uint32_t           capabilities; // tbd.

    char                *name;

    lifecycle_f         *calls;      // lifecycle entry points function pointers

    pod_properties      *prop;

/**
 * таблица функций ядра для драйвера - используется если нельзя пролинковать
 * напрямую (функции заменяются матросами, которые через эту таблицу идут в
 * ядро - см. pod_kernel_api.h)
 */
    kernel_f            *kernel_driver_api;

    void                *private; // private driver's data structure
}
```


};

Статическая линковка

Привязка статически линкованного драйвера к ядру должна осуществляться посредством ссылки из ядра на структуру дескриптора драйвера. Вопрос поддержки ядром списка доступных ему драйверов в этом документе не обсуждается.

Связка загружаемого драйвера

Для elf (pe) модулей - загрузить модуль, найти в таблице `get_pod_driver`, вызвать, получить дескриптор, работать через него. Если нет таблицы символов - считать, что `get_pod_driver` находится в начале сегмента кода. Если нельзя найти сегмент кода - искать маджик дескриптора (структуры `pod_driver`) по всему бинарному драйверу.

```
pod_driver * get_pod_driver(void);
```