OpenPOD drivers API

Release 0.1-0

Dmitry Zavalishin

Nov 29, 2019

Contents

1 Introduction **1**
1.1 Rationale 1
1.2 What's inside 2

2 Project navigation **3**
2.1 Directory tree 3

3 Driver structure **5**
3.1 Driver and device class 5
3.2 Driver descriptor 6
3.2.1 Driver methods 6
3.2.2 Driver life cycle 7
3.3 Device descriptor 8
3.3.1 Device methods 8
3.3.2 Device life cycle 9
3.3.3 Class specific interfaces 10

4 Examples **15**
4.1 Simplest driver skeleton 15
4.1.1 Driver descriptor 15
4.1.2 Device descriptor 16
4.1.3 Driver functions 17

5 Reference **19**

CHAPTER 1

Introduction

OpenPOD is a specification of a driver API, structure and lifecycle.

Aim of this specification is to provide small and alternative operating systems with common basis to share drivers around.

Source codes for the project (and for this book) can be found in OpenPOD GitHub repository.

1.1 Rationale

There are tens of operating system projects in the world. From tiny experimental works to medium size projects to big and big old ones with a bing community.

All of them need drivers. All of OS authors work on quite the same drivers from early start and for ever and ever.

It seems just natural for OS authors community to define a standard for a driver API to share work around and help each other.

This document is an attempt to start such effort.

Goal is to define API and driver structure that is:

- **Simple enough.** There were previous efforts to define portable driver API which required driver (even simplest one) to be unnecessarily complex. This specification lets driver author to start from a simple skeleton and make trivial driver in a day or so.
- **Extencible.** It must let complex and specific driver to have reach API into the kernel which does not limit driver designer too much and does not enforce him to come around and add non-standard entry points.
- **Support different device classes naturally.** Video and network drivers are really different and provide APIs that have quite nothing in common. But drivers in each class generally provide identical entry points.
- **Support runtime device addition and removal.** Driver must be able to inform kernel about status change and add or remove devices in run time.
- **Support drivers providing different types of devices.** PS2 driver exports both keyboard and mouse. Video card driver can provide sound device for an HDMI port, and so on.

- **Let driver to be both statically and dynamically linked.** Though, current specification is not really checked in a project with a dynamic driver loading, there is corresponding support in API and data structures.

Current state of specification: It is partially complete and waits for your comments.

The best way to comment is to create an issue in project's GitHub repository.

1.2 What's inside

This specification covers:

- Services provided to the driver by kernel
- Driver start / device discover / stop procedures (driver lifecycle)
- Device announce to the kernel (device lifecycle)
- Device API
- Configuration (PCI) information access
- Asyncronous IO requests

CHAPTER 2

Project navigation

2.1 Directory tree

GitHub project directory tree:

src Source code

> **openpod**
>
> > Headers.
>
> **libs**
>
> > Libraries of default functions and support subsystems for OpenPOD drivers.
>
> **os**
>
> > Example bindings for some operating systems.
>
> **test**
>
> > Regress test suite. Work in progress.
>
> **examples**
>
> > Code examples.
>
> **modules**
>
> > Loadable kernel modules. Mostly experimental.
>
> **barebones**
>
> > Driver skeletons: starting points to implement or port a driver.

dox Sources for this document,

CHAPTER 3

Driver structure

OpenPOD driver includes:

- **Driver descriptor.** A structure that defines global (not device specific) properties an entry points.

  Driver descriptor is a required part of a driver.

- **Device descriptor.** A structure which describes each device exported by driver. Can be single and static, or can be dynamically allocated in run time. For example, USB disk driver can create device descriptors dynamically as devices come and go.

  Device descriptor is not a required part - there can be driver which does not export devices at all or just starts with empty set of devices. But most drivers will provide at least one device at start.

- **Driver and/or device properties descriptors.** Set of key=value parameters to control driver or device and request meta information. For example, sound driver can set sampling rate with corresponding property.

  Properties are optional part of specification.

3.1 Driver and device class

Device class specifies API this device exports to the kernel.

```
#define POD_DEV_CLASS_VOID 0        // No API.
#define POD_DEV_CLASS_SPECIAL 1     // Has user defined nonstandard 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
```

Note: Sound driver class is missing.

Sound device is, basically a character (byte stream) device, but separate class can help kernel to classify and connect driver in a special way.

3.2 Driver descriptor

```
typedef struct pod_driver
{
    uint32_t            magic;

    uint8_t             API_version_major;
    uint8_t             API_version_minor;

    uint8_t             arch_major;
    uint8_t             arch_minor;

    uint8_t             class_id;
    uint8_t             pad0;
    uint8_t             pad1;
    uint8_t             state_flags;

    char                *name;

    pod_dev_required_f  calls;
    pod_dev_optional_f  *optional;

    pod_properties      *prop;

    void                *private_data;

    kernel_f            *kernel_driver_api;

} pod_driver;
```

All the fields of this structure will be described in reference part of this document, now we will just note the most interesting ones.

state_flags Used to track driver state, such as if driver found its hardware or not.

name Driver name, printable text for user to ideitify it.

calls Main set of driver entry points, see below.

private_data Private driver's state. Simple drivers keep here a link to an only device descriptor.

All the other fields can be zeroes or have default predefined values.

3.2.1 Driver methods

pod_construct Driver initialization. Driver must allocate structures and do any setup required but do not scan hardware or attempt to export devices. This stage can be empty.

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

pod_destruct Complete driver destruction. Driver must free all resources, stop threads, release OS objects (such as mutexes and conds). Driver can be unloaded after this call.

It is assumed that `deactivate` was called before this call. Nevertheless if no `deactivate` was called driver must attempt to do its best to deactivate self before desctruction. If it is not possible, this call can fail.

```
errno_t pod_destruct( pod_driver *drv )
```

pod_activate

Start driver. Prepare and activate hardware. Inform kernel about all devices this driver exports. See `pod_dev_link`.

This call is done before any IO is attempted by kernel. Until this call driver must refuse to do any IO.

Hardware scan must be done before this call.

```
errno_t pod_activate( pod_driver *drv )
```

pod_deactivate

Stop driver. Reset and stop hardware. Revoke all exported devices. See `pod_dev_unlink`.

```
errno_t pod_deactivate( pod_driver *drv )
```

pod_sense

Look for hardware to exist and be operational. Scan for devices that can be operated by this driver.

Devices must be found but not exported to kernel or started.

It is ok to call kernel entry points to scan hardware, such as PCI enumeration functions.

```
errno_t pod_sense( pod_driver *drv )
```

pod_probe

This is a call from kernel with an offer of a device to handle.

Driver must check if this device can be handled and accept or refuse to handle it. As with `pod_sense` accepted device should not be started or exported to kernel during processing of this call.

This entry point is not finally defined. Request for comments.

```
errno_t pod_probe( pod_driver *drv, bus?, dev? ) // ENOMEM, EFAULT
```

3.2.2 Driver life cycle

Defines driver initialization, start, search for hardware, stop and resource release stages.

Basic life cycle:

- Kernel calls `pod_construct`, driver allocates data strutures and resources (mutexes, threads, etc).
- Kernel calls `pod_sense` and/or `pod_probe`, driver looks for hardware and decides on list of devices that exist and can be served.
- Kernel calls `pod_activate`, driver exports to kernel known devices with calls to `pod_dev_link`.
- Driver serves IO requests. If devices come or go driver calls `pod_dev_link` / `pod_dev_unlink` to inform kernel.
- Kernel calls `pod_deactivate`, driver deregisters all devices.
- Kernel calls `pod_destruct`, driver releases all resources.

- Kernel can unload driver at this point.

3.3 Device descriptor

```
struct pod_device {
    uint32_t        magic
    uint8_t         class

    uint8_t         pad0
    uint8_t         pad1

    uint8_t         flags

    pod_driver      *drv
    pod_dev_f       *calls
    pod_properties  *prop
    void            *class_interface
    void            *private_data

    // 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
}
```

3.3.1 Device methods

pod_dev_stop

Stop device. Called after all IO is done and all possible users disconnected. In Unix like OS this entry point is called after device was closed by least process. No IO can be done after stop.

pod_dev_start

Start device. Called before first IO attempt on device.

In Unix like OS corresponds to open system call for this device.

```
errno_t pod_dev_stop( pod_device *dev )
errno_t pod_dev_start( pod_device *dev )
```

pod_rq_enqueue

Start asyncronous IO on device.

pod_rq_dequeue

Revoke IO request previously enqueued by `pod_rq_enqueue`, if possible. Can fail if IO is already in progress.

This entry point is optional.

pod_rq_fence

Make sure that all IO started before this call is finished. Can return instantly or wait for all previous IO is complete. In any case must guarantee that on call to request callback for this request all previously enqueued requests are complete and callbacks for them are finished too.

This entry point is optional.

pod_rq_raise

Change (usually - rise:) request priority.

Optional, but for disk IO it is really good thing to have.

```
errno_t pod_rq_enqueue( pod_device *dev, pod_request *rq )
errno_t pod_rq_dequeue( pod_device *dev, pod_request *rq )
errno_t pod_rq_fence( pod_device *dev, pod_request *rq )
errno_t pod_rq_raise( pod_device *dev, pod_request *rq, uint32_t io_prio )
```

3.3.2 Device life cycle

- Driver detects device, constructs descriptor and calls `pod_dev_link` for this device.
- Kernel connects device according to device class.
- Kernel decides to use device, calls `pod_dev_start` for it.
- Kernel calls sync interface methods or sends in IO requests for async IO. Device executes requests. See below for details.
- Kernel decides to cease using device, calls `pod_dev_stop`.

There are two possible interfaces that device can implement.

Asyncronous interface

Main entry point for this iterface is `pod_rq_enqueue` entry point. Kernel constructs `pod_request` structure and calls `pod_rq_enqueue`. Driver adds request to queue and returns immediately. As soon as request is processed by driver request is filled with results (status code) and `done` function is called. (Pointer to `done` function is in the request structure.)

OpenPOD project library provides helper functions for working with request queue.

There is also a proxy function (`pod_default_enqueue`) provided that converts asyncronous calls to syncronous ones, so for a simple driver just a set of sync entry points can be provided. We discourage writing drivers this way, but for a really fast tasks or alpha level drivers this is ok.

Here is an example of making asyncronous request to a video driver. (Well, it is meaningless to do this call in async style, but this example shows that just any call can be done in both sync and async style.)

```
pod_request *rq = calloc( 1, sizeof(pod_request) + sizeof(struct pod_video_rq_mode) );
if( rq == 0 ) { /* fail */ }

struct pod_video_rq_mode *rq_arg = ((void*)rq) + sizeof(pod_request);

rq->request_class    = POD_DEV_CLASS_VIDEO;
rq->operation        = pod_video_getmode;
rq->io_prio          = 0;
rq->op_arg           = rq_arg;
rq->done             = req_done_func;
```

(continues on next page)

(continued from previous page)

```
rc = pod_rq_enqueue( dev, rq );
if( rc ) { /* fail */ }
```

After completing request `done` function (`req_done_func()` in this example) will be called and `rq_arg` will be filled with requested data.

Warning: Callback can be called **before** `pod_rq_enqueue` return!

Don't rely on assumption that `pod_rq_enqueue` will return first and callback function called next. For long calls such as (non flash) disk read can beheave this way repeatedly, but it is **never** guaranteed.

For example, it is extremely wrong to write something like

```
rc = pod_rq_enqueue( dev, rq );
pod_kernel_lock_mutex( ... )
pod_kernel_wait_cond( ... )
pod_kernel_unlock_mutex( ... )
```

It is ok to do it this way:

```
pod_kernel_lock_mutex( ... )
rc = pod_rq_enqueue( dev, rq );
pod_kernel_wait_cond( ... )
pod_kernel_unlock_mutex( ... )
```

But signalling cond in `done` function must be guarded by the same mutex too.

Syncronous interface

Each device class can provide a set of methods that is specific for this class. An `class_interface` field of device descriptor points to an array of functions, which implement class specific operations. There are `pod_io_<CLASS_NAME>.h` headers defining class specific interfaces. Currently interfaces for block, network and video drivers are provided.

These methods are syncronous and return only if operation requested is done or error occured.

It is possible to set up proxy code that converts syncronous calls to asyncronous ones, so driver developer has to provide just one of them.

```
struct pod_video_rq_mode rq_arg;

rc = pod_dev_method( dev, pod_video_getmode, &rq_arg );
if( rc ) { /* fail */ }
```

3.3.3 Class specific interfaces

Both syncronous and asyncronous entry points implement same set of operations.

For async interface operation is selected by `request_class`, `operation` pair of request structure fields. Of course, request_class must be equal to target device's class.

Sync interface selects function by calling correct entry point. There is a helper function provided, but user code can do the same directly.

```
errno_t pod_dev_method( pod_device *dev, int op_id, void *param )
{
    if( dev == 0 ) return ENODEV;
    if( (op_id < 0) || (dev->class_interface == 0) ) return ENOSYS;

    errno_t (*class_func)(pod_device *dev, void *arg);

    // TODO check op id > max possible
    class_func = dev->class_interface[op_id];
    if( class_func == 0 ) return ENOSYS;

    return class_func( dev, param );
}
```

Video IO interface

Common types and structues.

```
typedef enum pod_video_operartions
{
    pod_video_nop,
    pod_video_getmode, pod_video_setmode, pod_video_getbestmode,
    pod_video_clear_all, pod_video_clear,
    pod_video_move,
    pod_video_write, pod_video_read,
    pod_video_write_part, pod_video_read_part
} pod_video_operartions;

// Pixel format, pod_pixel_rgba is preferred
typedef enum pod_pixel_fmt
{
    pod_pixel_rgba,     // 32 bit RGBA, A byte is ignored by HW
    pod_pixel_rgb,      // 24 bit RGB
    pod_pixel_r5g6b5,   // 16 bit, 5-6-5
    pod_pixel_r5g5b5,   // 16 bit, 5-5-5
} pod_pixel_fmt;

// flags
#define POD_VIDEO_IGNORE_Z  (1<<0)  // ignore z coordinate (but update z buffer)
#define POD_VIDEO_IGNORE_A  (1<<1)  // ignore A byte (aplha channel)
```

Clear:

```
struct pod_video_rq_sqare
{
    uint32_t    x, y;
    uint32_t    x_size, y_size;
};
```

Move from screen to screen:

```
struct pod_video_rq_2sqare
{
    uint32_t    from_x, from_y;
    uint32_t    from_x_size, from_y_size;

```

(continues on next page)

(continued from previous page)

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

Put or get complete bitmap:

```
// write, read
struct pod_video_rq_rw
{
    uint32_t    x, y;
    uint32_t    x_size, y_size;

    uint32_t    z;
    uint32_t    flags;

    char        *buf;

    pod_pixel_fmt   buf_fmt;
};
```

Put or get partial bitmap:

```
// write_part, read_part
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
    uint32_t    flags;

    char        *buf;

    pod_pixel_fmt   buf_fmt;
};
```

Parameters for video mode related ops:

```
// getmode, setmode, getbestmode
struct pod_video_rq_mode
{
    uint32_t    x_size, y_size;
    pod_pixel_fmt   buf_fmt;

    physaddr_t  vbuf;       //    0?    ?
};
```

Block IO interface

There are three possible operations defined for block IO devices.

```
enum pod_block_operartions
{
    pod_block_nop,
    pod_block_read, pod_block_write,
    pod_block_trim      // SSD specific
};
```

Parameters for `trim` operation:

```
struct pod_block_rq
{
    diskaddr_t  block_no;
    uint32_t    block_count;
    uint32_t    block_size;
};
```

Parameters for data IO (read, write):

```
struct pod_block_io_rq
{
    diskaddr_t  block_no;
    uint32_t    block_count;
    uint32_t    block_size;

    physaddr_t  physmem_addr;
};
```

Network IO interface

Network class specification is incomplete.

```
enum pod_video_operartions
{
    nop, get_mac_addr,
    mac_send, mac_recv,
    //ip_send, ip_recv,
};
```

Parameters for `get_mac_addr` operation:

```
#define POD_NET_MAX_MAC 16

// get_mac_addr
struct pod_net_rq_mac
{
    uint8_t mac_len;
    uint8_t mac[POD_NET_MAX_MAC];
};
```

CHAPTER 4

Examples

4.1 Simplest driver skeleton

lets review a simplest driver example provided in project repository.

Please see `barebones/simple` directory for this example's source code.

4.1.1 Driver descriptor

First we need to provide driver descriptor structure.

```
pod_driver simple_driver = {
    POD_DRIVER_MAGIC,
    POD_DRIVER_VERSION,
    POD_DRIVER_ARCH_UNKNOWN,

    POD_DEV_CLASS_VIDEO, 0, 0, 0,

    "simple driver skeleton - RENAME ME!",

    {
        simple_driver_construct,
        simple_driver_destruct,


        simple_driver_activate,
        simple_driver_deactivate,


        simple_driver_sense
    },

    0, // no optional entry points
```

(continues on next page)

(continued from previous page)

```
    0, // no properties

    0, // Private state does not exist yet

    0, // Statically linked, no need for kernel entry points struct

};
```

The only notable things here are:

- POD_DEV_CLASS_VIDEO - we declare that this is a video driver. OS might decide to start different drivers sooner or later, or to group drivers start by classes.
- simple_driver_X - entry points to start and stop our driver by OS.

4.1.2 Device descriptor

Our driver will provide just one and only device, so device descriptor can be static too.

```
pod_device simple_device =
{
    POD_DEVICE_MAGIC, // magic

    POD_DEV_CLASS_VIDEO,
    0, 0,
    0,       // flags

    // Device name, either static or runtime-generated
    "simple driver skeleton device name - RENAME ME!",

    &simple_driver,

    &dev_func,       // dev io entry points

    0,       // no properties

    0,       // dev class specific interface
    0,       // private data

    0,       // default request q
    0,       // request we do now
    0,       // thread used to run requests
    0,       // triggered to run next request

};
```

Again, we note that dev class is POD_DEV_CLASS_VIDEO. It is critical to be correct - OS will connect device to its users based on this.

We provide device entry points for async interface, but all of them are default ones provided by OpenPOD library.

```
static pod_dev_f dev_func =
{
    pod_default_enqueue,
    pod_default_dequeue,
```

(continues on next page)

(continued from previous page)

```
    pod_default_fence,
    pod_default_raise
};
```

First of them, `pod_default_enqueue`, will just call syncronous entry point of driver, three other are no-ops.

4.1.3 Driver functions

All the driver lifecycle functions are trivial.

Constructor and destructor (`simple_driver_construct` and `simple_driver_destruct`) are just empty - we don't need them for this driver to work.

simple_driver_activate

This function is mostly copied from system default implementation.

simple_driver_deactivate

Again, function is copied from system default implementation.

CHAPTER 5

Reference

This section is not ready yet.