Harman User Guide

Harman Device Virtualization for Connected Vehicles

Virtual driver writer's guide

Software Version 12.0

Doc. Rev. 2.2 / 528

Reference: DV-005

Date: 3/24/2022

HARMAN

Table of Contents

Table of Figures

1 Writing Virtual Drivers for HypervisorLinux Guest OS

The Hypervisor lets guest OSes run in Virtual Machines (VMs). Guest OSes and their applications need to access devices. Hence, it may happen that different VMs need to access concurrently the same device.

Devices may fall in one of the following categories:

- Physical devices managed by the Hypervisor and transparently shared by all VMs, e.g. timers
- Physical devices used and "owned" by a single VM. Some physical devices can interact transparently with several VMs, each of them using and owning a partition of the actual device. For example, some network controllers can serve up to eight VMs simultaneously.
- Physical devices shared by several VMs. For example, a disk controller may be used by several VMs. This case relies on a split-model driver. A VM owns the physical device and serves requests from other VMs to access the device. The VM owning the device runs a back-end virtual driver while the other VMs run a front-end virtual driver.
- Pure virtual devices. These are not physical devices and usually provide communication services between the different Guest OSes.

This guide covers the latter two categories.

Some virtual drivers are delivered with the **Device Virtualization for Connected Vehicles**product. Since they have been designed for a given version of the Linux kernel, it may be required to modify these virtual drivers to work seamlessly with a different Linux kernel version.

It may be needed to develop new virtual drivers to cover specific needs for a given platform.

This document explains how to write such virtual drivers. It starts with a simple example and then expands into more complex drivers.

A virtual driver runs inside a virtual machine, communicates with the Hypervisor and implements an API for the guest OS running in the VM, either Linux or a real-time OS. This guide mainly focuses on Linux virtual drivers. However, some information is provided for QNX users at the end of this document.

Linux virtual drivers are stored in the `src/vdrivers-3.18/drivers/vlx` directory of the work tree, independently of the Linux kernel version which they target. The actual location depends upon the delivery you received. "`vdrivers-3.18`" might be different in your file tree.

We assume the reader is familiar with how to write a Linux device driver, although we may explicit some interactions between a virtual device driver and the Linux kernel. We also assume the reader is familiar with the Hypervisor concepts.

The following chapters describe:

- A simple driver reacting to inter VM cross interrupts.
- The "paddle" driver enabling two VMs to exchange data in a ping-pong like manner. It builds on the cross interrupts mechanisms introduced by the first example.
- How to implement "lock-free ring buffers". This chapter is a generic introduction to this mechanism and has no specific details related to the Hypervisor. If you are familiar enough with the intricacies of such a mechanism, you may consider skipping this chapter.

- The virtual bidirectional pipe (aka vbpipe) driver that is an implementation of the lock-free ring-buffer described in the previous chapter.
- The virtual Remote Procedure Call (aka VRPC) driver which provides an RPC mechanism between two VMs.
- A comprehensive list of rules and guidelines to help you write a virtual driver.
- Some QNX-related considerations.

The source code for the cross interrupt, paddle and VRPC examples is provided along this document. The `vbpipe` virtual driver is an actual virtual driver part of the product delivery.

Virtual device drivers interact explicitly with the Hypervisor, for example to retrieve configuration information, to allocate some resource, to generate an event and so on. All the interactions are available within a library called NKDDI. Please refer to the NKDDI(3D) manual for details. These services are available from Guest OSes.

2 First Example: Reacting to Cross Interrupts

Here, we are going to create a simple device driver which:

- Initializes itself
- Prints some properties acquired from the Hypervisor
- Reacts to cross interrupts generated by other VMs and/or the Hypervisor
- Terminates gracefully

Such a virtual driver uses the Hypervisor APIs and the Linux APIs but does not communicate with other virtual machines.

Cross interrupts are events generated by the Hypervisor and delivered to Guest OSes as interrupts. The Hypervisor provides services enabling a Guest OS to send a given cross interrupt, but also to attach a handler to a cross interrupt.

2.1 Linux Driver Files

To create a virtual driver, one needs to edit at least three files inside the driver directory:

- The `Kconfig` file, which makes the driver visible in the Linux kernel configuration menu
- The `Makefile`, which contains compilation directives
- The driver source file, which contains the code of the driver

This is not different from what you would have to do for any Linux driver.

This driver is called `vexample1`, hence we add following code inside the `Kconfig` file:

```
config VLX_VEXAMPLE1
        tristate
        default y
config VEXAMPLE1
        tristate "Virtual driver example 1"
        default y
        select VLX_VEXAMPLE1
        help
          Virtual driver example 1.
```

```
          To compile this driver as a module, choose M here: the
          module will be called vexample1.
```

and the following code inside the `Makefile`:

```
# virtual example drivers
$(call vlx-vdriver, vexample1)
```

The `vlx-driver` macro is defined in the `Make-scripts.mk` file. Its discussion is beyond the scope of this document.

The source code is stored inside the `vexample1.c` file.

2.2 Driver Initialization

The driver interacts with the Hypervisor to:

- Get the identifier of its VM
- Get some other information
- Attach a routine to be executed whenever a "`SYSCONF`" cross interrupt is triggered in this VM
- Trigger such a cross interrupt

Please refer to the NKDDI(3D) manual to get detailed information about the corresponding Hypervisor services. All interactions with the Hypervisor are handled through the NKDDI layer and are named after the following pattern: `nkops.nk_xxx_yyy`. All other calls are related to Linux kernel. Some more detailed information is provided after the code.

Here is the code initializing the driver:

```
static NkXIrqId          vexample1_sysconf_xirqid;
static struct semaphore  vexample1_sem;

    static int
vexample1_module_init (void)
{
    const NkOsId my_id = nkops.nk_id_get();
    printk (VEXAMPLE1_INFO "starting, VM %d last VM %d running VMs 0x%x\n",
            my_id, nkops.nk_last_id_get(), nkops.nk_running_ids_get());
    printk (VEXAMPLE1_INFO "NK_XIRQ_SYSCONF %d, bit 10 gives mask 0x%x, "
            "LSBit in %x is %d\n", NK_XIRQ_SYSCONF, nkops.nk_bit2mask (10),
            1 << 9, nkops.nk_mask2bit (1 << 9));
    sema_init (&vexample1_sem, 0);    /* void */
    vexample1_sysconf_xirqid = nkops.nk_xirq_attach (NK_XIRQ_SYSCONF,
                                                     vexample1_xirq,
                                                     (void*) (long) my_id);
    if (!vexample1_sysconf_xirqid) {
        printk (VEXAMPLE1_ERR "xirq attach(%d) failed\n", NK_XIRQ_SYSCONF);
        vexample1_cleanup();
        return -EINVAL;
    }
    nkops.nk_xirq_trigger (NK_XIRQ_SYSCONF, my_id); /* void */
    down (&vexample1_sem);      /* void */
    printk (VEXAMPLE1_INFO "initialized\n");
    return 0;
}
module_init(vexample1_module_init);
```

Thanks to the special `module_init()` macro, `vexample1_module_init()` is the first function called by Linux driver framework. It initializes our driver. It first performs some stateless calls to

the NKDDI[1] library component (which wraps hypercalls and is typically compiled into the Linux kernel), and displays results:

- current virtual machine id, from `nkops.nk_id_get()`
- highest present virtual machine id, from `nkops.nk_last_id_get()`
- bitmask showing which virtual machines are currently running, from `nkops.nk_running_ids_get()`
- numerical value of the special SYSCONF cross interrupt (should be 512 or sometimes larger)
- the index of the least-significant bit set in 32-bit value, from `nkops.nk_mask2bit()`
- the mask corresponding to a bit, from `nkops.nk_bit2mask()`

2.3 Sending and Reacting to Cross Interrupts

Then the driver exercises the Hypervisor cross interrupt mechanism ("xirq" in short). To stay simple, it triggers the cross interrupt towards its own VM and waits until it arrives. In order to wait for the cross interrupt to happen without using CPU time, it uses a Linux kernel semaphore, with an initial counter of zero, so that the first wait is blocking. This semaphore is initialized using the `sema_init()` call.

It attaches a handler function to a predefined cross interrupt, called the system configuration cross interrupt, SYSCONF in short[2]. From this point on, the interrupt can happen at any time. The API is called `nkops.nk_xirq_attach()` and is mostly implemented inside the NKDDI library, though at some point the Hypervisor will be invoked to unmask the interrupt.

To make sure the interrupt happens, another Hypervisor API is used, which triggers an interrupt towards a virtual machine, be it another one or the current one: `nkops.nk_xirq_trigger()`. Unlike the `nkops.nk_xirq_attach()` call, which mostly happens inside the NKDDI library and inside the Linux kernel, triggering is mostly executed inside the Hypervisor.

Interrupt handlers may run concurrently with Linux base threads. To avoid race conditions between threads and interrupt handlers, interrupt handlers commonly wake up base threads by means of semaphores. Hence, in the interrupt handler, the semaphore is signaled using the Linux kernel `up()` primitive:

```
    static void
vexample1_xirq (void* cookie, NkXIrq xirq)
{
    printk (VEXAMPLE1_INFO "got xirq %d with cookie %p\n", xirq, cookie);
```

[1] Nanokernel (i.e. Hypervisor) Device Driver Interface

[2] Although SYSCONF is generally referenced through the NK_XIRQ_SYSCONF macro, this is not a constant, but a variable, exported from the NKDDI module. Historically, SYSCONF used to be cross interrupt 512, but nowadays the value can be higher, depending on the number of hardware interrupts managed by a given board.

```
    up (&vexample1_sem);
}
```

Then the entry routine sleeps on the semaphore waiting for the interrupt to happen, or consuming an interrupt which already happened, using the `down()` primitive

When run, the driver will produce messages like the following ones:

```
VEXAMPLE1: starting, VM 2 last VM 4 running VMs 0x1e
VEXAMPLE1: got xirq 512 with cookie 0000000000000002
VEXAMPLE1: initialized
VEXAMPLE1: got xirq 512 with cookie 0000000000000002
VEXAMPLE1: got xirq 512 with cookie 0000000000000002
```

The cookie value inside the handler is the same as the parameter passed to the `nkops.nk_xirq_attach()` call.

You might be surprised about getting multiple **`SYSCONF`** cross interrupts, when only one has been sent from the driver. This is because other entities in the system also generate them, typically in system boot or reconfiguration phases.

2.4 Unloading the Driver

Once the driver has initialized, the initialization thread goes away, while the driver code and data stay in memory, accepting **`SYSCONF`** invocations and printing traces. If the driver has been loaded as a module, it can be removed from memory using the `rmmod(1)` command-line utility or the `delete_module(2)` system call.

For this to not crash the Linux kernel at the next **`SYSCONF`** interrupt arrival, as the code will be removed from the Linux kernel virtual address space, the handler must be detached. This is accomplished with the cleanup code. This cleanup code can be also internally used for failed initializations:

```
    static void
vexample1_cleanup (void)
{
    if (vexample1_sysconf_xirqid) {
        nkops.nk_xirq_detach (vexample1_sysconf_xirqid);
    }
}

    static void
vexample1_module_exit (void)
{
    vexample1_cleanup();
    printk (VEXAMPLE1_INFO "module unloaded\n");
}
module_exit(vexample1_module_exit);
```

In our case, the module can be removed at any time, and the removal procedure is not supposed to fail, hence the "void" type for the `vexample1_module_exit()` routine.

To properly manage more complicated drivers, especially ones exporting services to the kernel itself, Linux maintains module reference counts, and will not invoke the module exit routine if the module is still busy from its point of view. In this case, `rmmod(1)` will immediately fail, though some options may be used to force unloading.

In other cases, objects allocated from the module and given to the kernel might have a reference count, and persist even after the unloading of the module, for example if they are currently being looked at by other code.

3 Second Example: Paddle Inter VM Communication

In this second example we create a vdriver to be inserted in two different VMs which enable these VMs to send data to each other: VM A generates data for VM B and tells VM B data is ready using a cross interrupt. VM B processes the data and replies similarly to VM A. Data will be allocated in a memory region shared by the two VMs.

The communication mechanism uses the vlink service provided by the Hypervisor.

3.1 Device Tree Vlinks Configuration

A virtual communication link (in brief vlink) is a peer-to-peer, unidirectional communication channel between a server-side end-point and a client-side end-point. Usually the vlink server and client sides are located in two different VMs but they could be located in the same VM.

In brief:

- vlink end-points must be described in the Hypervisor device tree at system build time
- Related end-points must use the same vlink "name"
- According to the configuration, each vlink end-point has a set of cross interrupts which can be sent to the VM owning that end-point, and a persistent memory region private to the VM
- Finally, a memory region that can be accessed by the two VMs is associated with the vlink and may be used to share data between these two VMs.

"Figure 3-1: Inside a vlink" illustrates a vlink managed by the Hypervisor. We explain the different resources progressively.

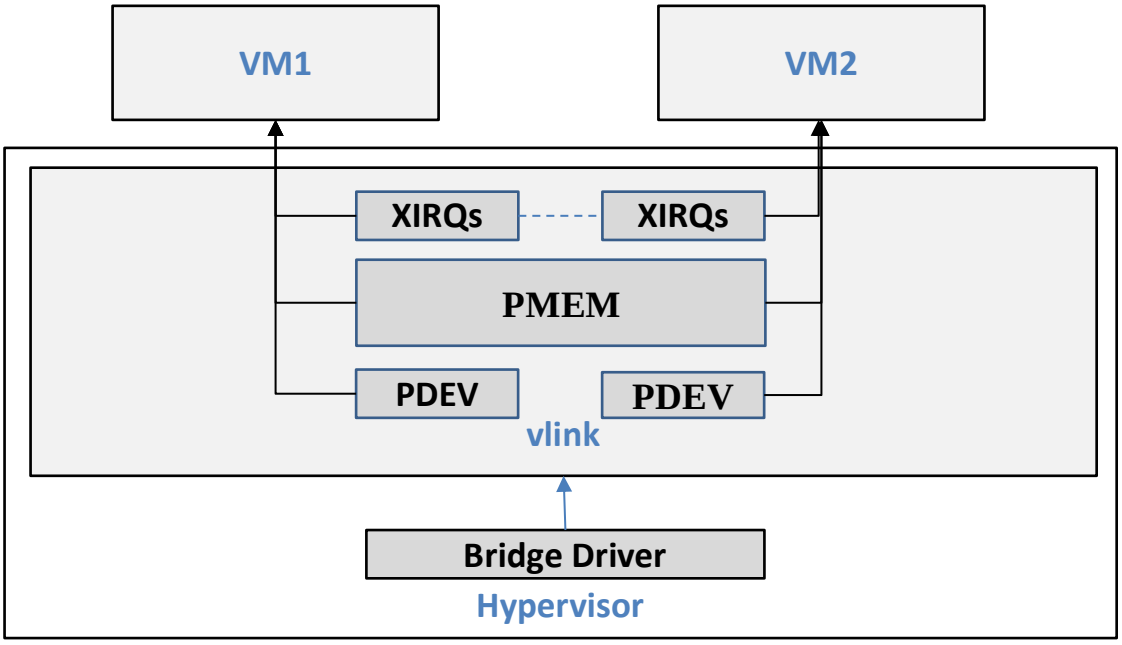


Figure 3-1: Inside a vlink

Each vlink connects two virtual machines, so memory can be only shared between 2 virtual machines at once.

If you are using the SADK board, the following text can be appended to the `src/nkernel-samsung/bsp/board/exynos8/sadk/vdt-vcar.dts` file, which defines the device tree for the SADK board Hypervisor configuration. This device tree is then passed to the build of the system image through one of the `TPL_VLM_DTB_FILE = $($(BOARD)_BSP)/vdt-vcar.dtb` lines in the `scripts/components/boards/sadk/vlm_def` file, depending on what configuration you have selected to build.

```
&vm2_vdevs {
    paddle_1: paddle@1 {
        compatible = "vexample2";
        server;                     // server end point
    };
};
&vm3_vdevs {
    paddle@2 {
        compatible = "vexample2";
        peer-phandle = <&paddle_5>;      // peer vlink
        client;                     // client end point
    };
};
&vm4_vdevs {
    paddle@4 {
        compatible = "vexample2";
        peer-phandle = <&paddle_1>;      // peer vlink
        client;                     // client end point
    };
    paddle_5: paddle@5 {
        compatible = "vexample2";
        server;                     // server end point
```

```
    };
};
```

In this device tree fragment, `&vm2_vdevs`, `&vm3_vdevs` and `&vm4_vdevs` are references to "node labels" defined elsewhere in the Hypervisor device tree. This allows attaching vlink definitions to existing Virtual Machine configuration.

To create a vlink, one must define two end-points. Such a definition is located inside per-VM sub-trees of the Hypervisor device tree. So, in order to match the two end-points of each vlink, the "node label" mechanism is used again. In the above example, `paddle_1` is the label of a vlink inside VM 2, and VM 4 then references this label using the `peer-phandle` property. Together, this creates a single vlink between VM 2 and VM 4.

Conventionally, the `peer-phandle` side should be marked as `client`, and the node-labeled side as `server`. Indeed, other ends might reference the same node label, creating additional vlinks.

The selection of the "paddle" name, which is arbitrary, is related to what the example vdriver is going to do, which is discussed later.

Similarly, VM 4 configuration references `paddle_5` label inside VM 3, which creates another vlink, between these two virtual machines.

The `client` and `server` are Boolean properties that allow drivers to distinguish between sides and behave differently. The `compatible` property names the vlink and allows drivers to find it in the Hypervisor database. Peer vlink end-points should have the same name, or the property could be omitted, in which case the name will be taken from node name, except for the `@` sign and whatever follows it to make it unique.

3.2 Paddle Vdriver Initialization

Here is the `module_init()` function of the "paddle" driver[3].

```
    static int
vexample2_module_init (void)
{
    const NkOsId myid = nkops.nk_id_get();
    NkPhAddr plink = 0;

    while ((plink = nkops.nk_vlink_lookup("vexample2", plink)) != 0) {
        NkDevVlink* vlink = (NkDevVlink*) nkops.nk_ptov(plink);
        BUG_ON (nkops.nk_vtop (vlink) != plink);
        if (vlink->s_id == myid) {
            vexample2_paddle_setup(plink, vlink, 1);
        }
```

[3] It is simplified here versus what is provided as example source code.

```
        if (vlink->c_id == myid) {
            vexample2_paddle_setup(plink, vlink, 0);
        }
    }
    return 0;
}
module_init (vexample2_module_init);
```

We can see a new API call, `nkops.nk_vlink_lookup()`, which allows to find the vlink end-points. Each vlink end-point is a data structure of type `NkDevVlink`. It is located in a memory region called **`PDEV`**, for persistent device memory. A vlink is a device rather than just memory, because some fields are updated by the Hypervisor.

"Figure 3-2: vlinks in PDEV memory regions" shows two VMs linked with a single vlink. IPA, for Intermediate Physical Addresses, designates what the virtual machine perceives as physical memory, PA stands for Physical Addresses and VA stands for Virtual Addresses.

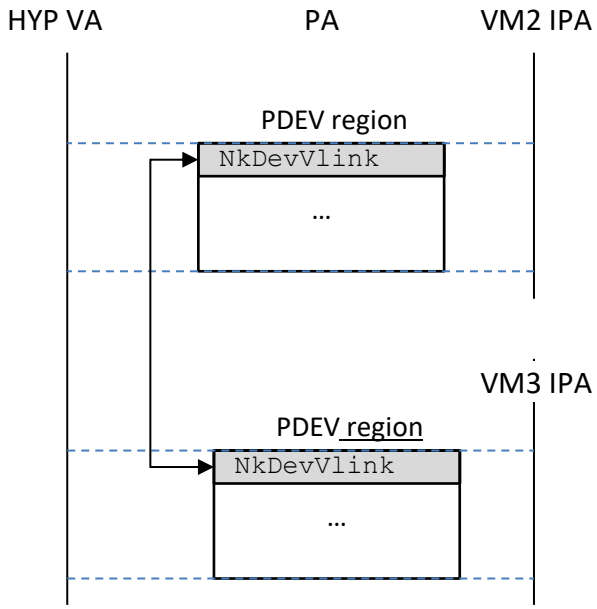


Figure 3-2: vlinks in PDEV memory regions

Passing address of one vlink to `nkops.nk_vlink_lookup()` returns the address of the next one, until there are no more. Passing address 0 gives the first vlink.[4] `NkPhAddr` is an integer type large

[4] `NkPhAddr` is an integer and not a pointer, so we do not initialize it with NULL.

enough to store the full range of physical addresses of the virtual machine. The "vexample2" string allows filtering the list; passing NULL would iterate on the full list of vlinks. This is the string used in the compatible device tree property.

Another NKDDI library call, nkops.nk_ptov() allows to convert this physical address to a virtual address. Indeed, at initialization time, the NKDDI library maps the entire **PDEV** region somewhere into the VM virtual address space and exports this call for access.

Note that this nkops.nk_ptov() call is not valid for any other physical address type and might panic the system if used with such an unexpected address.

We verify the result of nkops.nk_ptov() with the opposite nkops.nk_vtop() call, which should return the original value. Again, this call is only valid for memory located in the PDEV region. Note that the code line is there only to showcase the nkops.nk_vtop() call, so it can be freely removed.

Once the vlink is visible in virtual address space, the code examines two fields inside, vlink->s_id and vlink->c_id and compares them with the current VM identifier.

The s_id and c_id fields are initialized by Hypervisor from the client and server properties of the vlink nodes in device tree, taking into account which VM the vlink definition is located in. Therefore, if VM 2 contains a vlink node marked server and VM 4 contains a vlink node marked client, then the Hypervisor will create a vlink descriptor where s_id is 2 and c_id is 4.

Both VMs will see the same values. However, each VM gets its own vlink end-point descriptor. The Hypervisor is in charge of propagating changes from one end-point to the other and validating them in the process. The element in charge of this process is called a bridge. Such a bridge is illustrated on "Figure 3-1: Inside a vlink".

If the s_id field is equal to current VM id, then this VM is supposed to act as server, otherwise it acts as client. This information is passed as a Boolean to the second level initialization function, vexample2_paddle_setup(), which is described in the next chapter.

The example driver contains a function to display all the vlink fields:

```
    static void
vexample2_vlink_dump (const NkDevVlink* vlink)
{
    printk (VEXAMPLE2_INFO "@      : 0x%p\n",   vlink);
    printk (VEXAMPLE2_INFO "name   : %s\n",     vlink->name);
    printk (VEXAMPLE2_INFO "link   : %d\n",     vlink->link);
    printk (VEXAMPLE2_INFO "s_id   : %d\n",     vlink->s_id);
    printk (VEXAMPLE2_INFO "s_state: %d\n",     vlink->s_state);
    printk (VEXAMPLE2_INFO "s_info : 0x%lx\n",  (long) vlink->s_info);
    printk (VEXAMPLE2_INFO "c_id   : %d\n",     vlink->c_id);
    printk (VEXAMPLE2_INFO "c_state: %d\n",     vlink->c_state);
    printk (VEXAMPLE2_INFO "c_info : 0x%lx\n",  (long) vlink->c_info);
}
```

These other fields are described later. Right now, we can hint that name stores the vlink's name. This field is actually used by the nkops.nk_vlink_lookup() library routine to filter out unwanted vlinks, on top of a Hypervisor call which enumerates all of them.

Let's note that a single VM can be both a server and a client for the same vlink. This is why the module initialization function looks at both s_id and c_id. Both fields could be equal to myid. Such a vlink can be created with this device tree fragment:

```
&vm2_vdevs {
    paddle@1 {
        compatible = "vexample2";
    };
};
```

3.3 Vlink Usage

3.3.1 Local and Shared Memory

Here we are going to enter specific `paddle` vdriver initialization, related to what the driver is really going to do. As the "paddle" word might suggest, we are going to implement a ping-pong exchange of interrupts and data between virtual machines:

- A "paddle" is a local data structure, which is going to save all the state around one vlink. Each VM has its own. It is not very simple, so we do not show it here.
- A "ball" is the memory shared between virtual machines. It is much simpler, so we quote it here:

```
typedef struct {
    nku32_f count;
    nku32_f s_random;
    nku32_f c_random;
} vexample2_ball_t;
```

There are only a few fields, and they are defined using specific-width data types provided by Hypervisor interface include files (also called "nanokernel", hence the `nk` prefixes). This ensures that all VMs see the same layout in memory, even though some might be 32 and other 64 bits, and hence can interoperate at C language level, without marshaling and un-marshaling of data to some "portable network format".

Error management code present in the actual code is omitted here for the sake of clarity.

```
    static vexample2_paddle_t*
vexample2_paddle_setup (NkPhAddr plink, NkDevVlink* vlink, int server)
{
    vexample2_paddle_t* paddle;

    paddle = kzalloc(sizeof(*paddle), GFP_KERNEL);
    paddle->server = server;
    paddle->plink = plink;
    paddle->vlink = vlink;
    paddle->my_vmid = nkops.nk_id_get();
```

The above code allocates the local descriptor and fills it with some environment data. Usage of `kzalloc()` guarantees that memory is going to be zero-filled by default, so if we later add some fields to the `vexample2_paddle_t` type, they are going to have a stable initial value even if we forget explicit initialization.

3.3.2 Shared Memory Allocation

```
    paddle->pmem = nkops.nk_pmem_alloc(plink, 0, sizeof(*paddle->ball));
    paddle->ball = nkops.nk_mem_map(paddle->pmem, sizeof(*paddle->ball));
```

The above code allocates a chunk of shared memory between VMs and maps it inside the current Linux kernel address space. Shared memory is attached to our vlink, which is identified by its physical address `plink`. Given that there can be several chunks, each one is identified by an `id`, which should be a small integer; we selected value 0. Both VMs need to use the same `id` to allocate or access the same chunk.

The `nkops.nk_pmem_alloc()` call is idempotent: the first invocation allocates the chunk, and subsequent ones return its address. If two VMs use that call on the same vlink with the same resource id, only one chunk of memory is allocated by the first invocation. The returned address can be different in different virtual machines, though it is possible to configure the system so as it remains the same.

The chunk comes from a region of memory called Persistent Memory, or **PMEM** in short (hence the naming). Indeed, once allocated, it will survive till the reboot of the physical machine, even if both VMs are shut down or rebooted. Its content will also persist.

"Figure 3-3: PMEM memory shared through vlink" below shows a block of **PMEM** shared between 2 virtual machines. As explained earlier, IPA, for Intermediate Physical Addresses, designates what the virtual machine perceives as physical memory, PA stands for Physical Addresses and VA stands for Virtual Addresses.

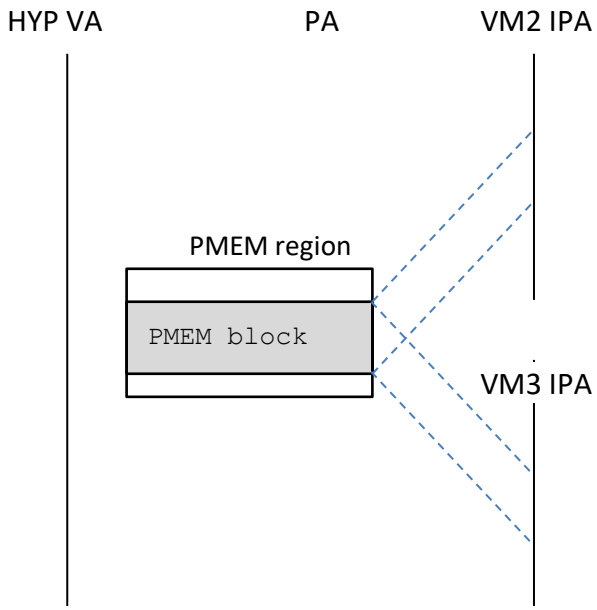


Figure 3-3: PMEM memory shared through vlink

Both calls can fail, so error management needs to deal with that. For example, a VM can only allocate a certain maximum (configurable) amount of PMEM. Similarly, mapping could fail if Linux runs out of virtual address space.

There is no "`nkops.nk_pmem_free()`" call, or similar. Once a persistent memory chunk is allocated, it stays so until the system reboots.

It is not necessary to use cache operations (`dmac_map_area()` calls on Linux) on the persistent memory when it is only shared between VMs. Linux cache operations will need to be used if shared memory is given to device drivers performing Direct Memory Access (DMA).

3.3.3 Asymmetrical Initialization

Up to now, the initialization work was the same for both sides of the vlink. However, because both VMs have their own vlink end-point descriptor to describe the same vlink, they need to use them differently, depending on which side they are on. For example, a peer VM id (to which we are going to send interrupts) is stored in:

- `c_id` if we are on server side
- `s_id` if we are on client side

Peers are also going to use the shared-memory ("ball") fields differently, depending on their role. This is expressed in the code below, with different initialization of the `my_random` and `peer_random` fields:

```
    if (server) {
        paddle->peer_vmid    = vlink->c_id;
        paddle->my_state     = &paddle->vlink->s_state;
        paddle->peer_state   = &paddle->vlink->c_state;
        paddle->my_irq_res   = 0;
        paddle->peer_irq_res = 1;
        paddle->my_random    = &paddle->ball->s_random;
        paddle->peer_random  = &paddle->ball->c_random;
    } else {
        paddle->peer_vmid    = vlink->s_id;
        paddle->my_state     = &paddle->vlink->c_state;
        paddle->peer_state   = &paddle->vlink->s_state;
        paddle->my_irq_res   = 1;
        paddle->peer_irq_res = 0;
        paddle->my_random    = &paddle->ball->c_random;
        paddle->peer_random  = &paddle->ball->s_random;
    }
```

The vlink descriptor contains state-type fields `s_state` and `c_state`, which allow to synchronize the initialization between VMs: each VM defines a certain state for itself and can also read the state of the peer. Hence the `my_state` and `peer_state` pointers, which always give the proper value for us and for peer. We will discuss the states later.

3.3.4 Interrupt Allocation

Once we modify something in shared memory, we need to send an interrupt to the peer, to wake it up and make it take corresponding action — the sole fact of writing into shared memory does not signal the peer in any way. Similarly, the peer needs to wake us up in opposite circumstances. Therefore, we need to have two interrupts:

- interrupt `#0` received by server and sent by client
- interrupt `#1` received by client and sent by server

A problem to solve is: how do we tell peer which interrupt is awaited by us, to signal changes on shared memory attached to the vlink? The Hypervisor helps with this: all cross interrupts are allocated through the vlink interface and referenced with arbitrary "local ids", on which server and client just need to agree in advance.

The interface is `nkops.nk_pxirq_alloc()`, and one needs to specify in which VM the interrupt is going to execute, to remove ambiguity and to obtain a number specific to that VM. Paddle driver decides to allocate `#0` for server side and `#1` for client side. This gives the following code:

```
paddle->my_irq =
 nkops.nk_pxirq_alloc(plink, paddle->my_irq_res, paddle->my_vmid, 1);
paddle->peer_irq =
 nkops.nk_pxirq_alloc(plink, paddle->peer_irq_res, paddle->peer_vmid, 1);
```

(Last parameter of 1 is a count and specifies that only 1 cross interrupt needs to be allocated for the `id` passed. Indeed, for some usages we might want to have a pool of sequentially numbered cross interrupts behind an `id`.)

The first invocation of these calls allocates the cross interrupts, and the subsequent ones just return previously assigned values. This is the same as with shared memory.

Note that `my_irq` and `peer_irq` can be the same value, because these interrupts are in different virtual machines. These values are not Linux-kernel numbers. They are relative to a virtual interrupt controller maintained for each VM by the Hypervisor.

Note that there is no "`nkops.nk_pxirq_free()`" call, or similar. Once a persistent cross interrupt number is allocated, it stays so until system reboots.

3.3.5 Attaching to Cross Interrupt

A handler needs to be attached to the local interrupt, just like in example 1:

```
paddle->irqid = nkops.nk_xirq_attach(paddle->my_irq, vexample2_hdl,
                                     paddle);
```

From this point on, the `vexample2_hdl()` handler can be executed immediately.

```
    static void
vexample2_hdl (void* cookie, NkXIrq xirq)
{
    vexample2_paddle_t* paddle = cookie;
    if (*paddle->my_state == NK_DEV_VLINK_ON) {
        queue_delayed_work(system_wq, &paddle->struck_work, HZ);
    }
}
```

We can see that handler accesses the `struck_work` field, which was not initialized yet. This is not a defect, and the access is not going to happen yet, because `my_state` is still in the default `NK_DEV_VLINK_OFF` condition. States of vlinks and how to use them to solve initialization and shutdown coordination issues are covered in detail in the next paragraph.

The interrupt handler just makes a Linux kernel thread (`system_wq`) execute a base-level handler of ours, after HZ clock ticks[5] or 1 second from interrupt occurrence. This time delay simulates the flight of the ball from peer to us. In the base-level handler, we are going to send the ball back.

We need to initialize the delayed work Linux structure and associate our own handler to it:

```
    INIT_DELAYED_WORK(&paddle->struck_work, vexample2_struck);
```

Here is the handler, the base-level routine that is "sending back the ball":

```
    static void
vexample2_struck (struct work_struct* work)
{
    vexample2_paddle_t* paddle = container_of (work, vexample2_paddle_t,
                                               struck_work.work);
    printk (VEXAMPLE2_INFO "(%s-%d-%s) count=%d\n", paddle->vlink->name,
            paddle->vlink->link, paddle->server ? "server" : "client",
            paddle->ball->count);
    ++paddle->ball->count;
    nkops.nk_xirq_trigger (paddle->peer_irq, paddle->peer_vmid);
}
```

Out of a pointer to the `work_struct` field passed by Linux kernel, it figures out the container (*paddle*), using the `container_of()` macro, which subtracts a small value from pointer and typecasts the result to our local type. This allows finding the `count` counter in shared memory and increment it, signaling ball reception to peer.

We use the same `nkops.nk_xirq_trigger()` call as in example 1 to send a cross interrupt to peer to signal this change. But this time, the second parameter indicates peer VM rather than our own VM.

3.4 Vlink Handshake Mechanism

The above code could pretty much do the ping-pong work by its own (if we remove the state test from `vexample2_hdl()`). We just need to trigger a first cross interrupt to one of the VMs, and it will send it back to peer, which in turn will trigger a reaction, which will trigger another reaction of ours, etc. Each time a 1 second delay mentioned previously is applied, to avoid an interrupt flood in both VMs.

However, such operation would remain unmanaged and partially chaotic. We cannot tell if the peer is there and if we should expect an answer to our ping. A lack-of-response timeout would need to be introduced to detect very simple problems.

[5] The Linux kernel does not use periodic timer interrupts at HZ frequency anymore, but still offers APIs that are expressed in clock ticks rather than micro or milliseconds.

This is where the vlink handshake mechanism can help: two fields of the vlink descriptor named `s_state` and `c_state` can be used to synchronize the two peers using the vlink. They have an enumerated type and can have values `OFF` (default), `RESET` (on going) and `ON` (up).

Each driver manages its own field and watches the peer's field. Each change of these states is signaled by **SYSCONF** interrupt. This cross interrupt does not need to be allocated; it is predefined and global for all vdrivers. Hence, handlers may be invoked when changes happen for other vlinks than their own; such invocations should be ignored.

The three possible states for a peer are the following ones:

- `OFF`: there is no driver loaded for vlink
- `RESET`: the driver has performed local initialization and expects peer to reach the same state
- `ON`: driver has noticed peer is either in **RESET** or **ON** state, so it sets itself as **ON**. Both peers become **ON**.

A driver will therefore set its state to **RESET** when loading and watch for peer to reach the same state or **ON**. Then it can enter **ON** state.

Conversely, to break communications, a driver sets its state to **OFF**. Peer will notice loss of **ON** and recycle itself as well.

This gives the following diagram and code. On initialization, arrows start from mutual **OFF** state (left top) state and progress in parallel towards mutual **ON** state (right bottom). On breaking, arrows start from mutual **ON** state, towards mutual **OFF** state, skipping the **RESET** state.

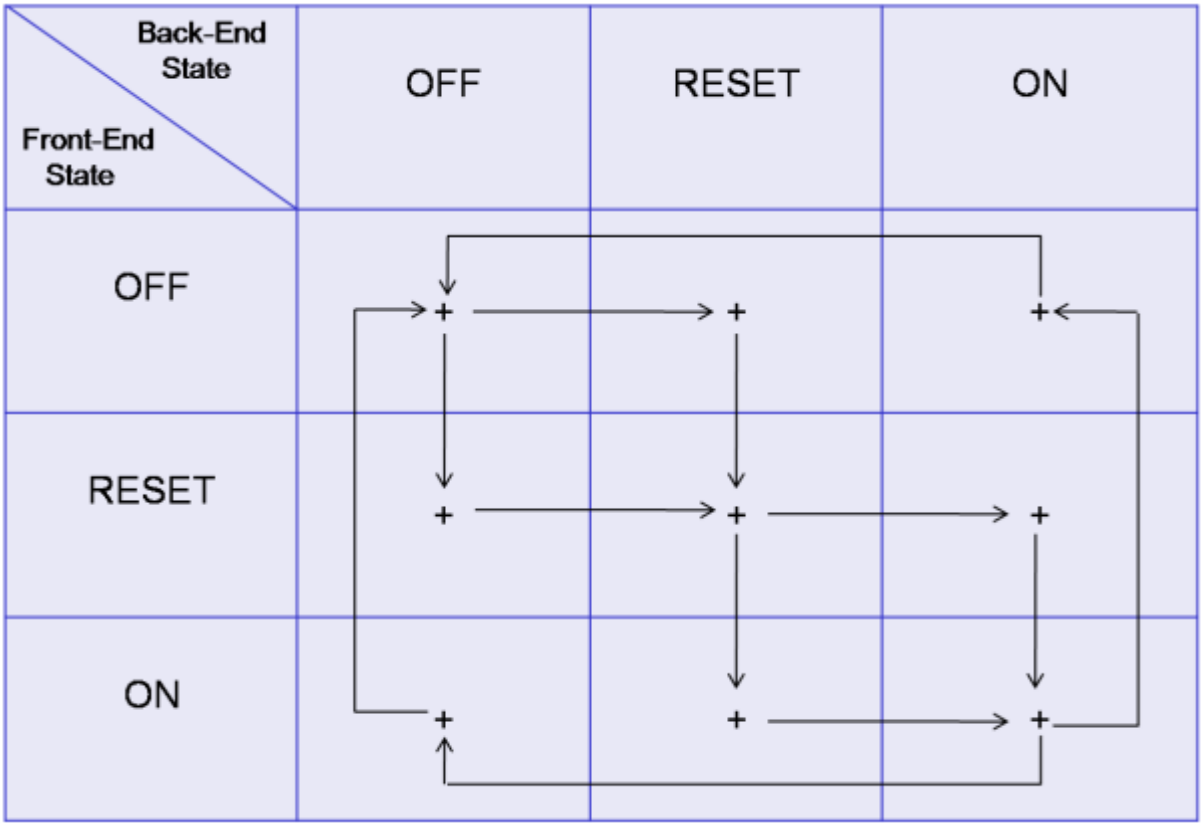


Figure 3-4: vlink end-point state transitions

3.5 Vlink Handshake in Practice

```
    static void
vexample2_handshake (struct work_struct* work)
{
    vexample2_paddle_t* paddle = container_of (work, vexample2_paddle_t,
                                               handshake_work);
    switch (*paddle->my_state) {
    case NK_DEV_VLINK_OFF:
        if (*paddle-> peer_state != NK_DEV_VLINK_ON) {
            vexample2_game_init (paddle);
            *paddle->my_state = NK_DEV_VLINK_RESET;
            nkops.nk_xirq_trigger (NK_XIRQ_SYSCONF, paddle->peer_vmid);
        }
        break;
    case NK_DEV_VLINK_RESET:
        if (*paddle-> peer_state != NK_DEV_VLINK_OFF) {
            *paddle->my_state = NK_DEV_VLINK_ON;
            nkops.nk_xirq_trigger (NK_XIRQ_SYSCONF, paddle->peer_vmid);
            vexample2_game_start (paddle);
        }
        break;
    case NK_DEV_VLINK_ON:
        if (*paddle-> peer_state == NK_DEV_VLINK_OFF) {
            *paddle->my_state = NK_DEV_VLINK_OFF;
            nkops.nk_xirq_trigger (NK_XIRQ_SYSCONF, paddle->peer_vmid);
            vexample2_game_stop (paddle);
        }
        break;
    }
}
```

Some comments about transitions:

- If we are **OFF**, and peer is **ON**, this means that we used to have a connection, but peer did not yet notice our resetting of it, hence we must wait until it does (by putting itself to **OFF** or **RESET** state). Therefore, we must wait for next **SYSCONF** interrupt and do nothing in between.
- If we are **RESET**, we should not enter **ON** state if the peer is **OFF**, because this means the peer is still initializing or still recycling the previous connection.
- If we are **ON**, there is only a loss of communications if driver is **OFF**. Otherwise, it could still not notice our previous transition from **RESET** to **ON** (following its own entry into **RESET** state).

These transitions can be performed from interrupt handlers attached to the **SYSCONF** interrupt but can also be deferred to thread level.

Specific actions are performed on specific transitions, and such actions need a thread context:

- **OFF** to **RESET**: `vexample2_game_init()`
- **RESET** to **ON**: `vexample2_game_start()`
- **ON** to **OFF**: `vexample2_game_stop()`

Instead of assuming the burden of creating a thread which is going to run only for very short periods of time, example 2 uses the Linux "work" mechanism.

```
    INIT_WORK(&paddle->handshake_work, vexample2_handshake);
```

This code above initializes the `work_struct` and associates a handler function with it.

```
paddle->sysconfid =
    nkops.nk_xirq_attach(NK_XIRQ_SYSCONF, vexample2_sysconf_hdl, paddle);
```

This code above attaches a **SYSCONF** cross interrupt handler function and implicitly unmasks the cross interrupt. This handler is trivial and just wakes up a thread:

```
    static void
vexample2_sysconf_hdl (void* cookie, NkXIrq xirq)
{
    vexample2_paddle_t* paddle = cookie;
    queue_work(system_wq, &paddle->handshake_work);
}
```

It is necessary to start the handshake by triggering a work execution. This can be either performed by self-sending of the **SYSCONF** interrupt, or by using a specific kernel API:

```
    queue_work(system_wq, &paddle->handshake_work);
```

Invoking the cross interrupt handler or the work routine directly from module initialization function is not correct, as proper context of execution and proper serialization cannot be guaranteed.

3.6 The Game

To start the game, the ball needs to be thrown for the first time, by sending a cross interrupt to peer. To better simulate the reality of a ping-pong game, only 1 ball should be thrown. Trivially, one of the parties (server or client) could always have this role. But it is more interesting to use a simple distributed negotiation, involving shared memory, to decide who is going to start. Each party initializes a random variable in shared memory before the game starts:

```
    static void
vexample2_game_init (vexample2_paddle_t* paddle)
{
    *paddle->my_random = get_random_int();
}
```

In order to decide who throws the ball, both values are summed, and the `server` Boolean is added too. This Boolean value is different on both sides, so the sum will be different too. The party that obtains the odd value (least-significant bit of the result set) throws the ball:

```
    static void
vexample2_game_start (vexample2_paddle_t* paddle)
{
    if ((paddle->server + *paddle->my_random + *paddle->peer_random) & 0x1) {
        queue_delayed_work(system_wq, &paddle->struck_work, HZ);
    }
}
```

Note that the `my_random` and `peer_random` values are initialized during the **OFF** to **RESET** vlink state transition, and they are only read after both vlinks sides reach the **RESET** state and transition to **ON** state. This guarantees that each virtual machine will see the correct stable peer value.

To stop the game, the "work" is cancelled:

```
    static void
vexample2_game_stop (vexample2_paddle_t* paddle)
```

```
{
    cancel_delayed_work_sync(&paddle->struck_work);
}
```

Let's note that when the driver code is unloaded, all the "works" also need to be cancelled.

3.7 Exit and Cleanup

Below is the teardown code for Paddle vdrivers, which allows to showcase the `nkops.nk_mem_unmap()` call.

```
    static void
vexample2_paddle_teardown (vexample2_paddle_t* paddle)
{
    if (paddle->sysconfid) {
        nkops.nk_xirq_detach (paddle->sysconfid);
    }
    if (paddle->irqid) {
        nkops.nk_xirq_detach (paddle->irqid);
    }
    if (paddle->struck_work_inited) {
        cancel_delayed_work_sync (&paddle->struck_work);
    }
    if (paddle->handshake_work_inited) {
        cancel_work_sync (&paddle->handshake_work);
    }
    if (paddle->my_state) {
        *paddle->my_state = NK_DEV_VLINK_OFF;
        nkops.nk_xirq_trigger (NK_XIRQ_SYSCONF, paddle->peer_vmid);
    }
    if (paddle->ball) {
        nkops.nk_mem_unmap (paddle->ball, paddle->pmem, sizeof(*paddle->ball));
    }
    kfree (paddle);
}
```

It can be invoked at any time during initialization, or for final cleanup.

First, cross interrupts are detached, to isolate code from external stimuli, then thread-level activities are stopped, to quiet down the code and data, and finally memory is released.

We should also inform the peer that we are gone but setting our end of the vlink to **OFF** state and by sending a **SYSCONF** cross interrupt to it. Otherwise, it might remain waiting for the ball to come back indefinitely, holding some resources unnecessarily. The **vexample2_game_stop()** routine will be executed in peer on vlink breakage.

Another reason to set the vlink state to **OFF** is to make it possible to reload the code later. If we left the vlink as **ON**, a future initialization would not proceed properly (please check the code of `vexample2_handshake()` to see why), because it expects an initial **OFF** state.

We could also clear the vlink state to **OFF** at start time in `vexample2_module_init()`, before starting the hand-shake. This helps running the same code in environments (typically QNX) where virtual drivers are POSIX processes, which are freely killable and hence cannot always perform cleanups. The operating system will release process resources, performing operations like detaching interrupt handlers and freeing memory, but it will not perform cleanups at Hypervisor level.

Not represented here is the `vexample2_module_exit()` function, which scans the list of paddles, which are singly-linked using a "next" field, invoking the above teardown routine for each

of them. The code makes sure paddle memory is not released before the "next" field inside has been saved to find the next paddle.

4 Lock-free Ring Buffers

This chapter describes how to implement a lock-free circular buffer. Nothing specific to the Hypervisor is introduced. Therefore, if you are familiar with such communication mechanism you may skip this chapter.

4.1 Principles

Lock-free ring buffers, also called circular buffers, can be used to communicate between virtual drivers, that is passing memory content around in an organized way.

Lock-free means there is no shared lock such as a mutex on which VMs would blindly, to implement a critical section, and no atomic operations, which are often implemented using assembly loops repeated an unpredictable number of times. Instead, virtual drivers send each other cross interrupts and use raw shared memory to synchronize.

A ring has a:

- *Write index*, updated by writer and readable by reader
- *Read index*, updated by reader and readable by writer

A ring has a predefined number of entries, typically sized 1 byte for pipes, or more, e.g. for Ethernet frames, possibly segmented.

We suppose here that the writer and reader only write and read 1 entry at a time. Writing or reading larger chunks leads to partial write and partial read situations, which require slightly more complicated processing. Such processing happens with `vbpipes`, described in "The Vbpipe Bidirectional Pipe Vdriver" chapter later in this document.

Both indexes can go from 0 till number of entries (`ring_size`) minus one, and then wrap around back to 0.

Special conditions are:

- Empty: `write_index == read_index`
- Full: `write_index == read_index`

This leads to ambiguity, which needs to be resolved somehow.

4.2 Full Versus Empty State Implementation

One possible way of telling apart "empty" and "full" ring situations is to sacrifice one of the ring slots. Therefore, a full condition would naturally happen when the `write_index` reaches `read_index` minus one. This is slightly wasteful in memory and does not really lead to simpler processing than the optimized algorithm described below.

To solve the full versus empty ambiguity, we introduce `full_flag` and `empty_flag` Booleans, also shared between reader and writer, which leads to the following algorithms:

- On write:

- check `full_flag` and only proceed if false.
- `++write_index`, wrap around if necessary
- if `write_index == read_index` then `full_flag=true`
- always set `empty_flag=false`

- On read:
  - check `empty_flag` and only proceed if false
  - `++read_index`, wrap around if necessary
  - if `read_index == write_index` then `empty_flag=true`
  - always set `full_flag=false`

This can be simplified, as follows.

One can remove both flags and use non-truncated indexes instead, i.e. indexes which progress freely, rather than being immediately wrapped to 0 when reaching ring_`size`. The `full` and `empty` status will be indicated by the most significant bits of the indexes. Of course, such extended indexes must be truncated to access the actual buffer for data reading and data writing:

- Empty: write_index == read_index
- Full: write_index == read_index + ring_size
- To deal with overflows, we use index differences:
  - empty: `write_index - read_index == 0`
  - half: `write_index - read_index == ring_size/2`
  - full: `write_index - read_index == ring_size`

Eventually, both indexes will reach the maximum value and wrap over. They must be of unsigned integer type, or of `int` type specifically, or the difference must be truncated to their base integer type, otherwise sign extension problems will appear.

Also, because the high-order bits in indexes are now used as flags, the ring size can only be half of the maximum value of the integer type used for indexing.

4.3 Algorithm with Free-running Indexes

The above gives the following algorithm:

- On write:
  - check (`write_index - read_index < ring_size`) and refuse write otherwise
  - write at `write_index % size`
  - `++write_index`
- On read:
  - check (`write_index - read_index` != 0) and refuse read otherwise
  - read from `read_index % size`
  - `++read_index`

4.4 Simple Signaling

We need to deal with situations where the ring is empty or full, and put the reader or writer to sleep, respectively, using operating system mechanisms, so as not to waste CPU time, which is shared among all VMs.

- On write, we must wake up reader if ring becomes *non-empty*
- On read, we must wake up writer if ring becomes *non-full*

Of course, we could wait a little before wakeup, in order to give more space to writer, or more data for reader to take in one go. However, this is an optimization, which requires some prior knowledge about the rate of transferred data and is not part of the communication algorithm.

In the virtualized context, this wakeup can only be done through cross interrupts or "xirqs". Peers must agree which cross interrupt is used for what purpose. This is not part of this description.

We introduce 2 Boolean variables, `was_empty` and `was_full`. They are purely local and not shared between peers.

Below is the signaling version of the algorithm:

- On write:
  - `was_empty = (write_index == read_index)`
  - check `(write_index - read_index < ring_size)`, `++write_index` if not full, otherwise sleep or error
  - if `was_empty`, send *can-receive* cross interrupt
- Read:
  - `was_full = (write_index - read_index) == ring_size`
  - check (`write_index - read_index` != 0), `++read_index`, otherwise sleep or error
  - if `was_full`, send *can-transmit* cross interrupt

4.5 Optimal Signaling

Though it frees CPU when there are no data for reading and no space for writing, the above algorithm is still not optimal:

- The *can-receive* xirq is sent even if reader does not sleep yet
- The *can-transmit* xirq is sent even if writer does not wait yet

This leads to useless, though harmless, cross-VM interrupts.

A solution is to add two variables to shared memory:

- Introduce "*stopped"* or "*waiting"* flags for each side. We will call them `write_waiting` and `read_waiting`
- Only send xirq if flag set
- To avoid races with wakeup code, the *"waiting"* flag must be set *before* testing condition about the necessity to wait.

These variables are similar to condition variables or Linux `set_current_state (TASK_INTERRUPTIBLE)` and `schedule()` calls.

4.6 Almost Final Algorithm

On write:

- `was_empty = (write_index == read_index)`
- `write_waiting = true`
- check (`write_index - read_index < ring_size`) to see if space for write exists

- if full, sleep for *can-transmit* xirq, and repeat the above on wakeup
- `write_waiting = false`
- `++write_index`
- if `was_empty`, and `read_waiting`, send *can-receive* cross interrupt

On read:

- `was_full = (write_index - read_index) == ring_size`
- `read_waiting = true`
- check (`write_index - read_index != 0`) to see if something is available
- if empty, sleep for *can-receive* xirq, and repeat the above on wakeup
- `read_waiting = false`
- `++read_index`
- if `was_full`, and `write_waiting`, send *can-transmit* cross interrupt

Note that even if the writer, or the reader, does not plan to sleep for space/data immediately, they must set `write_waiting` and `read_waiting` respectively before checking the status of shared memory. This ensures that the cross interrupt will be delivered by peer on availability change.

4.7 Final Algorithm with SMP Support

4.7.1 Metadata Coherence

The above algorithm is still not entirely correct, because of SMP systems idiosyncrasies, which require specific remedies.

On SMP systems, each peer can access the various variables in parallel:

- `write_index`
- `read_index`
- `write_waiting`
- `read_waiting`

Hence, they may not be totally coherent, given performance-increasing hardware optimizations, such as out of order execution and cache consistency protocols.

Some values changed locally might appear changed in the peer sooner than others. There is no guarantee that the order of writes is always respected.

Yet, the `write_waiting` and `read_waiting` variables must be made visible to peer before we check the status of the shared memory through the other 2 variables, so that peer can trigger the cross interrupt appropriately.

If the reading (resp. writing) peer does not see the most up to date version of `write_waiting` (resp. `read_waiting`), and it perceives the ring as empty (esp. full) and goes to sleep, the other peer might fill (resp. empty) the ring without sending a notification xirq, leading to an infinite wait.

To overcome propagation issues, the Memory Barrier mechanism needs to be used.

A Write Memory Barrier must be set between the setting of the `write_waiting` and `read_waiting` variables, and the reading of indexes.

The peer needs to use a Read Memory Barrier before accessing the variables to get the most up to date values in the proper order.

Memory barriers have to be used in pairs for proper operation.

Note that using operating system sleeping primitives will automatically insert memory barriers.

4.7.2 Code

This gives the following final algorithm, with the 2 pairs of memory barriers, numbered 1 and 2.

On write:

- `was_empty = (write_index == read_index)`
- `write_waiting = true`
- **write memory barrier (#1)**
- check (`write_index - read_index < ring_size`) to see if space for write exists
- if full, sleep for *can-transmit* xirq, and repeat the above on wakeup
- `write_waiting = false`
- `++write_index`
- **read memory barrier (#2)**
- if `was_empty`, and `read_waiting`, send *can-receive* cross interrupt

On read:

- `was_full = (write_index - read_index) == ring_size`
- `read_waiting = true`
- **write memory barrier (#2)**
- check (`write_index - read_index != 0`) to see if something is available
- if empty, sleep for *can-receive* xirq, and repeat the above on wakeup
- `read_waiting = false`
- `++read_index`
- **read memory barrier (#1)**
- if `was_full`, and `write_waiting`, send *can-transmit* cross interrupt

4.7.3 Data Coherence

The same applies to data themselves, which are also part of shared memory and subject to same SMP artefacts.

Before incrementing the `write_index`, the writer needs to make sure that the buffer of data is entirely visible to peer. A write memory barrier needs to be added between data write to shared memory and index incrementation. Since memory barriers are used in pairs, the reader needs thecorresponding read memory barrier between the read of the write_index, to determine the amount of data available, and the read of the shared buffer.

Without this barrier, the CPU would be allowed to speculate the loads on the buffer before we check whether there are some data actually available in the pipe. By the time we determine that we need to read N bytes from the shared memory, the processor will happily use the result of the speculated loads yielding whatever stale data were present in memory at that time.

Conversely, before incrementing `read_index`, the reader must entirely read the provided buffer, to avoid the writer from overwriting it with new data. A memory barrier needs to be added between data reading from shared memory and the incrementing. On its side, the writer will check read_index to determine whether there is some room left in the shared buffer before writing it.

The barrier pairing rule should once again prompt for great caution here as the algorithm would not behave correctly if these two operations were reordered. The main difference with the previous case of the memory barriers around the use of write_index is that this time the memory operations dependent on the check of read_index are stores and stores cannot be speculated. There is no need for an explicit memory barrier in this case.

5 The Vbpipe Bidirectional Pipe Vdriver

The `vbpipe(4D)` driver[6] delivered with the Device Virtualization for Connected Vehicles leverages the lock-free ring buffers described in the previous chapter. Though there are two other "pipe" drivers part of the Device Virtualization Software delivery, the unidirectional one, called `vpipe(4D)`, and the memory-mapped one, called `vupipe(4D)`, `vbpipe(4D)` is by far the most used.

We first describe the external view (principles and Linux interface) and then discuss the implementation.

5.1 Vbpipe Description

5.1.1 Principles

vbpipes offer byte-oriented, bidirectional communication between VMs. Message-oriented communication can be implemented on top of vbpipes by prefixing data chunks with size, and possibly a tag, leading to the tag-length-value (TLV) paradigm.

The driver creates `/dev/vbpipeX` devices (default name pattern, where X stands for the minor number) in both VMs. Here are two example devices, corresponding to two independent vbpipes:

```
crw-rw---- 1 root root 223, 0 Jan 1 00:00 /dev/vbpipe0
crw-rw---- 1 root root 223, 1 Jan 1 00:00 /dev/vbpipe1
```

[6] The (4D) notation gives the location of the manual page, in chapter 4 subsection D (for device drivers).

The standard Linux pipe API can be used. It consists of `open(2)` (including the `O_NONBLOCK` option), `read(2)`, `write(2)`, `close(2)`, `ioctl(2)` (for `FIONREAD`), `select(2)`, `pselect(2)` and `poll(2)` system calls.

A vbpipe is symmetrical. There is no difference between the reading and writing ends and they behave independently. A vbpipe has a configurable buffer size, 4KB by default and 16 bytes minimum. There are two separate buffers of the same size, one for each direction. This size is defined within the Hypervisor device tree.

In the Linux kernel, the `vbpipe` vdriver is enabled by including the `CONFIG_VBPIPE` and `CONFIG_VLX_VBPIPE` options and by adding vlink nodes to the Hypervisor device tree, one per end-point.

5.1.2 Configuration

Below are sample vlink nodes for the Hypervisor device tree:

```
&vm2_vdevs {
    vbpipe23: vbpipe@0 {
        compatible = "vbpipe";
    };
};
&vm3_vdevs {
    vbpipe@0 {
        peer-phandle = <&vbpipe23>;
    };
};
```

They allow establishing a single vbpipe, between VMs 2 and 3, hence the "vbpipe23" label naming which allows to match ends.

These vlink definitions differ from those used in the previous chapters: they omit the `server` and `client` keywords. This has the effect of creating two, rather than just one vlink end-point descriptor in **PDEV**. This allows having separate states in both directions.

One of these paired vlinks is called the server vlink, while the other is called the client vlink. We will discuss later how the name is related to operations.

To create more vbpipe links, just add more device tree nodes , named differently than `vbpipe@0` (because nodes having the same name are combined).

Within the Hypervisor device tree, a vlink node can hold an *info* property which is a string. This string can be retrieved by the Guest OS. The format of the *info* property is defined by the Guest OS. For vbpipe, the vlink node *info* property has the following format:

```
info = "[a][;[size][;minor][;[wakeup=<msecs>][;name=<dev-name>]]]"
```

where:

- `a` = wait-only-on-empty read policy
- `size` = size in bytes of the shared memory area allocated to transfer data. Syntaxes valueK and valueM are also possible. Two buffers of `size` bytes are allocated.
- `minor` = required Linux device minor number
- `msecs` = number of milliseconds during which a wake-lock should be held following reception of data[7]
- `dev-name` = a customized device name (default is /dev/vbpipeX)

5.1.3 Linux Interface

By default, the device nodes in the filesystem are named `/dev/vbpipeX` and their major device number is usually 223.

The minor device number *X* depends on the vlink position in the VM device tree. It does neither depend on the "unit name" in device tree (the part of name after the `@` sign), nor on the vlink link id (stored in the `link` field of the `NkDevVlink` structure). It can be forced by setting the `name` field of the vlink `info` property (cf. full syntax in the previous section of this document).

Reads are fully-satisfied, they always return the requested number of bytes, with 2 exceptions:

- if flag `a` (*wait-only-on-empty*) is defined in *info* property
- if the producer side closed, in which case remaining data are returned even if partial

This allows writing simple code, which does not need to perform several partial reads to acquire a full message.

Non-blocking writes are also atomic, as long as message's lengths are smaller or equal to the shared buffer size.

Possible `write(2)` errors:

- **`EPIPE`** lost connection: the vbpipe is broken
- **`EINTR`** a signal interrupted the write
- **`EAGAIN`** if non-blocking write cannot be done
- **`EFAULT`** if the passed buffer is not accessible

Possible `read(2)` errors:

[7] A *wake-lock* is a Linux mechanism which permits to prevent the system from entering a power-save state for a given amount of time. This lets a task to execute for a certain time without being disturbed by power-save policies.

- **EAGAIN** if no more data and non-blocking
- **EINTR** a signal interrupter the read
- **EFAULT** if the passed buffer is not accessible

Only a close followed by re-open can repair a lost vbpipe, assuming the peer VM is still there.

5.1.4 Observation Through /proc/nk/vbpipe

The `/proc/nk/vbpipe` file allows observation and access to statistics. Example content:

```
    Mi  Pr  Id EaU Stat  Size  Opns  Reads  ReadBytes- Wrtes  WriteBytes
     0   2   0 E.0 FFFF   ff0     0      0           0      0           0
```

where:

- `Mi` = minor in Linux filesystem
- `Pr` = peer Virtual Machine id
- `Id` = vlink's unique `link` id, as there can be several vbpipes between a pair of virtual machines
- `EaU` =
  - "E" stands for Enabled (fully OK)
  - "a" stands for "a" flag passed
  - "U" is current open count, usually 0 or 1, though it can be higher if several processes read or write from vbpipe.
- `Stat` = state of the 2 vlinks and their 2 sides each:
  - client vlink, client side
  - client vlink, server side
  - server vlink, client side
  - server vlink, server side
  - F means **OFF**, R means **RESET** and O means **ON**
- `Opns` = total number of first-time opens, that is transitions from Closed to Open

Remaining columns have self-explanatory names.

5.1.5 Limitations

Currently, vbpipes have the following well-known limitations:

- `FIONREAD` does not care about write end being closed
- `open(2)` is always blocking, the `O_NONBLOCK` flag only applies to subsequent I/O operations
- `O_NONBLOCK` only applies to the first thread that performs a given operation, read or write. If a thread attempts non-blocking I/O while another thread is already blocked, it is blocked until the first thread unblocks.
- There is no waiting in `open(2)` if the vbpipe is dead, yet still open by other users

These limitations do not seem to be problematic for current users.

5.2 Implementation

5.2.1 Introduction

Vbpipes allow lock-free synchronization between VMs.

The implementation uses two vlinks for every vbpipe, and has to match them at initialization time. This is explained in "5.2.2 Matching Bidirectional Vlinks" chapter.

As described in the chapter "4 Lock-free Ring Buffers", there is one circular buffer (ring) per direction, two cross interrupts (*can-receive* and *can-transmit*), to signal *data-now-available* and *no-more-full* conditions, xirqs are sent only if peer is waiting. Memory barriers are used.

Vlinks remain **OFF** and xirq handlers detached until the vbpipe is open. Vlink state transitions are done from thread, not from xirq handlers, as shown in the "paddle" example. xirqs are sent as late as possible, to increase the likelihood of a higher-priority peer doing a full read.

Inside each VM, operations are performed in mutual exclusion. There are separate mutexes for read, write and open operations.

5.2.2 Vbpipe Vdriver Initialization

5.2.2.1 Entry Point

The entry point is identical to other virtual drivers:

```
    static int
vbpipe_module_init (void)
{
    /* ... */
    return 0;
}
module_init(vbpipe_module_init);
```

5.2.2.2 Counting Vbpipes from Hypervisor Configuration

The code manages as many vbpipes as configured in the device tree. Therefore, the initialization routine has to count all vlinks to service, in order to allocate appropriate memory resources. The result is stored inside `ex_dev_num`:

```
    NkPhAddr    plink;
    NkOsId      my_id = nkops.nk_id_get();

    ex_dev_num = 0;
    plink      = 0;
    while ((plink = nkops.nk_vlink_lookup("vbpipe", plink))) {
        vlink = (NkDevVlink*) nkops.nk_ptov(plink);
        if (vlink->s_id == my_id) {
            ex_dev_num += 1;
        }
    }
```

Note that only server side vlinks named "vbpipe" are counted. This is the effect of using bidirectional vlinks.

If there are no vlinks to manage, the initialization stops there.

The `ex_` prefix on some variables or routines is a leftover from legacy code.

5.2.2.3 Exporting a Linux Character Device Interface

The driver exports a Linux character device interface. This embeds two different notions:

- A character device-type node in the filesystem and associated callbacks
- Associated device objects inside the Linux kernel

The first action is accomplished using code below, indicating the desired major device number. If this number is already occupied, by some other device driver, the operation will fail, and we ask the Linux kernel to allocate a random major automatically instead. Having a random major number does not really matter much, because the peripheral is primarily referenced through its pathname.

```
    if ((ret = register_chrdev(VBPIPE_MAJOR, "vbpipe", &ex_fops))) {
        ex_major = register_chrdev(0, "vbpipe", &ex_fops);
```

The `ex_fops` data structure, passed as a parameter to Linux character device framework, defines the operations it provides:

```
static const struct file_operations ex_fops = {
    .owner   = THIS_MODULE,
    .open    = ex_open,
    .read    = ex_read,
    .write   = ex_write,
    .release = ex_release,
    .poll    = ex_poll,
    .llseek  = no_llseek,
    .unlocked_ioctl = ex_ioctl,
};
```

We do not provide a `.compat_ioctl` function as we do not have 32 bit ioctls (handling structures with 32 bit fields) to support on 64 bit systems.

A device class needs to be created inside the kernel. This way we will benefit from the standard management of devices, like `uevent` messaging, which will automatically populate the `/dev` filesystem each time we allocate a minor device, and visibility inside the `/sys` filesystem.

```
    static struct class* ex_class;
    ex_class = class_create(THIS_MODULE, "vbpipe");
```

5.2.2.4 Initialization of Vbpipe Device Descriptors

We rescan the vlinks to find the parameters for each one. Notably, given that the Linux minor number can be selected arbitrarily, we create a single array of vbpipe descriptors, directly indexed by the Linux device minor. This often requires increasing the previously computed `ex_dev_num` value:

```
    while ((plink = nkops.nk_vlink_lookup("vbpipe", plink))) {
        vlink = (NkDevVlink*) nkops.nk_ptov(plink);
        if (vlink->s_id == my_id) {
            unsigned required_minor;
            if (ex_param(vlink, NULL, 1, &required_minor) &&
                required_minor >= ex_dev_num) {
                ex_dev_num = required_minor + 1;
            }
        }
    }
```

We can now allocate memory for all the device descriptors:

```
    ex_devs = (ExDev*) kzalloc(sizeof(ExDev) * ex_dev_num, GFP_KERNEL);
```

The most important fields of `ExDev` are listed below.

```
#define WAIT_QUEUE    wait_queue_head_t
#define MUTEX         struct mutex

typedef struct ExDev {
    _Bool         enabled;       /* flag: device has all resources allocated */
    _Bool         defined;       /* this array entry is defined */
    NkDevVlink*   s_link;        /* server link */
    ExRing*       s_ring;        /* server circular ring */
    size_t        s_size;        /* size of server circular ring */
    size_t        s_pos;         /* reading position inside ring */
    NkXIrq        s_s_xirq;      /* server xirq for server ring */
    NkXIrq        s_c_xirq;      /* client xirq for server ring */
    NkXIrqId      s_s_xid;       /* server cross interrupt handler id */
    NkDevVlink*   c_link;        /* client link */
    ExRing*       c_ring;        /* client circular ring */
    size_t        c_size;        /* size of client circular ring */
    size_t        c_pos;         /* writing position inside ring */
    NkXIrq        c_s_xirq;      /* server xirq for client ring */
    NkXIrq        c_c_xirq;      /* client xirq for client ring */
    NkXIrqId      c_c_xid;       /* client cross interrupt handler id */
    MUTEX         olock;         /* mutual exclusion lock for open/release ops */
    MUTEX         rlock;         /* mutual exclusion lock for write ops */
    MUTEX         wlock;         /* mutual exclusion lock for read ops */
    WAIT_QUEUE    wait;          /* waiting queue for all ops */
    int           count;         /* usage counter */
    unsigned int flags;          /* device behavior semantic */
        /* Statistics */
    unsigned      opens;
    unsigned      reads;
    unsigned      writes;
    unsigned long long read_bytes;
    unsigned long long written_bytes;
    unsigned int  wakeup;        /* wakeup time-out in msecs/jiffies */
                                 /* (0 - disabled) */
    struct wakeup_source* ws;    /* wakeup source */
    char         name[16];
} ExDev;
```

We then scan the vlinks a third time and initialize the `ex_devs` entries with configuration information; this code is not quoted here. Notably, we attribute minors to devices for which they were not specified explicitly. We also save the vlink pointer with each device:

```
    ex_devs [required_minor].s_link = vlink;
```

Given that we can have holes in the `ex_devs` array, we mark really used entries with the `defined` Boolean field. Another Boolean, named `enabled` will signal entries which we managed to initialize till the end and which are fully operational.

5.2.3 Matching Bidirectional Vlinks

The vbpipe driver uses one vlink per communication channel, which means that:

- one vlink is used to manage the **reader to writer** connection between two VMs
- the other vlink is used to manage the opposite, **writer to reader** connection for the same couple of VMs

This means that each driver manages two `NkDevVlink` objects.

As introduced in section "5.1.1 Principles" above, one of the vlink end-points is called the server vlink end-point, and the other is called the client vlink end-point:

- the server vlink end-point has the `s_id` field matching the current VM
- the client vlink end-point has the `c_id` field in this role.
- both vlink end-points have the same `link` field value

Because each VM has a different id, the vdriver will naturally find out what is its role for each vlink end-point. As a result, the code can be identical for both sides: client and server.

There is no specific support for matching this couple of vlink end-points in the Hypervisor or in the NKDDI library. Hence the code in the vdriver:

```
    for (i = 0, ex_dev = ex_devs; i < ex_dev_num; i++, ex_dev++) {
        NkDevVlink* slink = ex_dev->s_link;
        plink             = 0;

        if (!ex_dev->defined) continue;
        while ((plink = nkops.nk_vlink_lookup("vbpipe", plink))) {
            vlink = (NkDevVlink*) nkops.nk_ptov(plink);
            if ((vlink != slink)       &&
                (vlink->c_id == slink->s_id) &&
                (vlink->s_id == slink->c_id) &&
                (vlink->link == slink->link)) {

                ex_dev->c_link = vlink;
                ex_param(vlink, ex_dev, 0, NULL);
                ex_dev_init(ex_dev, i);
                break;
            }
        }
    }
```

Error management has been omitted for brevity, as elsewhere.

Once this double loop has completed, every `ex_dev` has both the `s_link` and the `c_link` pointer fields set.

Note that if server and client vlink end-points are in the same VM, defining a loopback vbpipe, we have 2 identical vlink end-point objects:

- same `s_id`
- same `c_id`
- same `link` number

However, their `NkDevVlink` descriptors each have a different **PDEV**[8] memory address. In this case, we consider that the client vlink is the second vlink enumerated by the `nkops.nk_vlink_lookup()` function. Because of that we have the:

```
    vlink != slink
```

condition in the `if` statement above, which otherwise would not be required.

5.2.4 Initializing a Single Vbpipe

The `ex_dev_init()` function is called to initialize a device (vbpipe) instance during driver initialization phase, from `vbpipe_module_init()`.

```
    static void
ex_dev_init (ExDev* ex_dev, unsigned minor)
{
    NkDevVlink* slink = ex_dev->s_link;
    NkDevVlink* clink = ex_dev->c_link;
    struct device* cls_dev;
    . . .
```

It sets the device name, using the `vbpipe` prefix and minor, except if some custom name has been selected using the vlink `info` field:

```
    if (!strlen(ex_dev->name))
        snprintf(ex_dev->name, sizeof(ex_dev->name), "vbpipe%u", minor);
```

It allocates memory for the two communication rings:

```
  if ((ex_dev_ring_alloc(clink, &ex_dev->c_ring, ex_dev->c_size) != 0) ||
      (ex_dev_ring_alloc(slink, &ex_dev->s_ring, ex_dev->s_size) != 0)
```

where the initialization code does **PMEM** operations already seen in previous examples:

```
    static int
ex_dev_ring_alloc(NkDevVlink* vlink, ExRing** p_ring, int size)
{
    NkPhAddr plink;
    NkPhAddr pring;
    int      sz         = size + sizeof(ExRing) - MIN_RING_SIZE;

    plink = nkops.nk_vtop(vlink);
    pring = nkops.nk_pmem_alloc(plink, 0, sz);
    *p_ring = (ExRing*) nkops.nk_mem_map(pring, sz);
    return 0;
}
```

[8] It should have a higher address, but not necessarily, as the exact ordering of devices in **PDEV** is left to the Hypervisor implementation.

By convention, a driver connected to the client side of the communication link will write characters into the ring, and a driver connected to the server side of the communication link will read characters from the ring. This is coherent with the idea that a client sends a request first, to which a server reply.

In the same way we refer to the corresponding circular rings as the client ring, ring used by the client link, and as the server ring, ring used by the server link.

Each ring needs two cross interrupts (*can-receive* and *can-transmit,* seen in chapter "Lock-free Ring Buffers"), so we have four interrupts in total to allocate. Two of them will be only received (`c_c_xirq` and `s_s_xirq`) and two of them only sent (`c_s_xirq` and `s_c_xirq`):

```
    if ((ex_dev_xirq_alloc(clink, &ex_dev->c_c_xirq, 0) != 0) ||
        (ex_dev_xirq_alloc(clink, &ex_dev->c_s_xirq, 1) != 0) ||
        (ex_dev_xirq_alloc(slink, &ex_dev->s_c_xirq, 0) != 0) ||
        (ex_dev_xirq_alloc(slink, &ex_dev->s_s_xirq, 1) != 0)
```

Each individual allocation function checks whether we allocate the *can-receive* (data in Ring) or the *can-transmit* (space in Ring) interrupt, which have different slots in vlink xirq table.

```
    static int
ex_dev_xirq_alloc(NkDevVlink* vlink, NkXIrq* p_xirq, _Bool server)
{
    NkPhAddr plink;
    NkXIrq   xirq;
    plink = nkops.nk_vtop(vlink);
    if (server) {
        xirq = nkops.nk_pxirq_alloc(plink, 1, vlink->s_id, 1);
    } else {
        xirq = nkops.nk_pxirq_alloc(plink, 0, vlink->c_id, 1);
    }
    *p_xirq = xirq;
    return 0;
}
```

Function of cross interrupt	**Field name**	**Vlink p_xirq slot**	**Handler attached**
Server *can-receive*	`s_c_xirq`	0	No
Server *can-transmit*	`s_s_xirq`	1	Yes
Client *can-receive*	`c_c_xirq`	0	Yes
Client *can-transmit*	`c_s_xirq`	1	No

We do not attach the two cross interrupt handlers yet. This is done only when a `/dev/vbpipeX` device is opened.[9]

In order for Linux to manage the `uevents` and `sysfs`, we need to use a special API:

```
    cls_dev = device_create(ex_class, NULL, MKDEV(ex_major, minor),
                            NULL, ex_dev->name);
```

Following this call, the udev daemon (or equivalent user-space program monitoring uevents) will create the `/dev/xxx` file in filesystem, where `xxx` is defined by `ex_dev->name`.

Initialize mutual exclusion locks and the Linux kernel waiting queue:

```
    mutex_init (&ex_dev->olock);
    mutex_init (&ex_dev->rlock);
    mutex_init (&ex_dev->wlock);
    init_waitqueue_head(&ex_dev->wait);
```

Let us note that there is only 1 waiting queue for all 3 blocking operations on the vbpipe. It will be used:

- initially at open time
- later at read and write times

One of the reasons is that a given wait operation can be signaled from two different sources. For example, a read-operation wait, caused by lack of data in ring, can be awaked by:

- peer giving some data: this is the *can-receive* interrupt
- peer being rebooted or closing the pipe: this is the **SYSCONF** interrupt

Having different waiting queues would mean that the **SYSCONF** interrupt must awake the appropriate queue, or all of them at the same time.

Another reason is that a vbpipe end is usually serviced by a single thread, which only sleeps for one event at a time, either a read or a write, so there is no performance problem of awaking both readers and writers at the same time, if any.

Finally, because of Android context, where the system goes to sleep (Suspend to RAM) as soon as possible, to save on battery power, vbpipe supports preventing this after reception of data for a configurable number of milliseconds, for example 250 milliseconds, using the `wakeup_source` mechanism, which has to be initialized if the user decided to rely on it (by providing appropriate "info" field content in vlink definition):

[9] Or some other name selected by "info" field of vlink.

```
    if (ex_dev->wakeup) {
        ex_dev->ws = wakeup_source_register(ex_dev->name);
    }
```

After performing all the initializations, we can attach the **SYSCONF** handler:

```
ex_sysconf_id = nkops.nk_xirq_attach(NK_XIRQ_SYSCONF, ex_sysconf_hdl, 0);
```

5.2.5 Management of /proc/nk/vbpipe File

After the initialization for each vbpipe, we can establish a `/proc/nk/vbpipe` file which allows seeing the current status and collected statistics. The data available through this file is generated at open time and buffered inside Linux kernel until the file is completely read. To that end, we use the `struct seq_file` API.

Let's note that Linux only allows to buffer 4 KB of data this way.

```
    static int
ex_seq_proc_show (struct seq_file* seq, void* v)
{
    seq_printf (seq, ". . .\n");
}
    static int
ex_proc_open (struct inode* inode, struct file* file)
{
    return single_open (file, ex_seq_proc_show, 0);
}
static const struct file_operations ex_proc_fops =
{
    .owner      = THIS_MODULE,
    .open       = ex_proc_open,
    .read       = seq_read,
    .llseek     = seq_lseek,
    .release    = single_release,
};
```

The `struct file_operations` data type is used, as for character devices, but this time, all but one functions are provided by Linux kernel.

We attach these functions to a file in filesystem as follows:

```
static _Bool    ex_proc_exists;

    static int __init
ex_proc_init (void)
{
    struct proc_dir_entry* ent;
    ent = proc_create_data("nk/vbpipe", 0, NULL, &ex_proc_fops, 0);
    if (!ent) {
        printk (KERN_ERR "Could not create /proc/nk/vbpipe\n");
        return -ENOMEM;
    }
    ex_proc_exists = 1;
    return 0;
}
    static void
ex_proc_exit (void)
{
```

```
    if (ex_proc_exists) {
        remove_proc_entry ("nk/vbpipe", NULL);
    }
}
```

The `/proc/nk` directory is created by the NKDDI library, together with the `/proc/nk/id` file, so we can directly create the `vbpipe` node underneath.

We do not need the `struct proc_dir_entry` object pointer to delete the file at cleanup time, so we use a simple Boolean to signal that such a cleanup is required.

5.2.6 Opening a Vbpipe

5.2.6.1 External Entry Point

The `ex_open()` callback, which we previously registered through the `struct file_operations`, implements the open operation required by Linux semantics for character devices, and returns 0 if everything went fine:

```
    static int
ex_open (struct inode* inode, struct file* file)
{
    /* . . . */
    return 0;
}
```

The opposite operation is the `ex_release()` callback.

```
    static int
ex_release (struct inode* inode, struct file* file)
{
    /* . . . */
    return 0;
}
```

At the first open, vbpipe attaches the cross interrupt handlers and performs a vlink handshake (as previously exposed in "3.4 Vlink Handshake Mechanism") with the peer driver. At the last release, the ongoing communications are aborted.

5.2.6.2 Checking Minor Number and Device Validity

We need to check whether the device minor number is supported by the driver, and convert it into a local `ExDev` pointer:

```
    unsigned int minor   = iminor(inode);
    ExDev*       ex_dev;
    if (minor >= ex_dev_num) {
        return -ENXIO;
    }
    ex_dev = &ex_devs[minor];
```

Though Linux allocates the valid `/dev/vbpipeX` devices automatically, using the `uevent` and `udev` mechanisms, nothing prevents the super user from creating a device node by hand, using the `mknod(1)` utility, with any minor (or major) device number. In such a case, only the above check will prevent a kernel crash.

Given that our minor number space can be sparse, if some vbpipes have been configured with specific minors, we also need to check whether a given minor is fully initialized. For that purpose, we have the `defined` and `enabled` Booleans, `enabled` being stronger:

```
    if (ex_dev->enabled == 0) {
        return -ENXIO;
    }
```

5.2.6.3 Serializing Concurrent Opens

Linux allows devices and files to be opened several times in parallel. Each open causes the invocation of the `ex_open()` callback, which decides what to do with this specific instance.

Conversely, for every past open, the `ex_release()` callback is eventually invoked as well.[10]

The standard Linux framework ensures that the driver code cannot be unloaded while some minors are open.

Therefore, a driver must manage a reference count for every minor device and release resources only when this reference count drops to zero.[11]

Of course, a driver can also simply forbid multiple opens, but vbpipe manages this concurrency.

The reference count variable is called `count`. Parallel opens are serialized by the `olock` mutex. The first thread to open the vbpipe takes the lock and performs the handshake with peer, while other openers will wait until this process is finished, by trying to lock the same mutex. The mutex does not only protect the "count" variable, but also the whole open process:

```
    if (mutex_lock_interruptible(&ex_dev->olock)) {
        return -EINTR;
    }
    ex_dev->count += 1;
    if (ex_dev->count != 1) {
        mutex_unlock(&ex_dev->olock);
        return 0;
    }
    /* . . . */
    mutex_unlock(&ex_dev->olock);
    return 0;
```

Let's note that the above code does not respect the `O_NONBLOCK` flag: opening is always blocking, there is no refusal to open if this would require some waiting, and no asynchronous background open either. This is a well-known limitation. However, blocking is interruptible.

---

[10] Except in case of brutal reboot where processes were not individually killed beforehand.
[11] Some other UNIX or UNIX-like operating systems have an approach where the release function is only called on last close.

5.2.6.4 Attaching Cross Interrupt Handlers

As we remember from earlier sections, cross interrupts are not attached before device open, except for the shared **SYSCONF** cross interrupt.

The driver must attach the *can-receive* and *can-transmit* interrupt handlers for the appropriate directions. This is accomplished using the `ex_xirq_attach` helper subroutine:

```
    if ((ex_xirq_attach(ex_dev, ex_server_xirq_hdl, 1) != 0) ||
        (ex_xirq_attach(ex_dev, ex_client_xirq_hdl, 0) != 0)) {
```

The `server` parameter tells the helper which cross interrupt is used.

```
    static int
ex_xirq_attach (ExDev* ex_dev, NkXIrqHandler hdl, _Bool server)
{
    NkDevVlink* link;
    NkXIrq      xirq;
    NkXIrqId    xid;

    if (server) {
        link = ex_dev->s_link;
        xirq = ex_dev->s_s_xirq;
    } else {
        link = ex_dev->c_link;
        xirq = ex_dev->c_c_xirq;
    }
    xid = nkops.nk_xirq_attach(xirq, hdl, ex_dev);
    if (server) {
        ex_dev->s_s_xid = xid;
    } else {
        ex_dev->c_c_xid = xid;
    }
    return 0;
}
```

The cross interrupt handlers are almost trivial, as they just wake up the wait queue head of the device. The available data, room and device state will be retested by awaken threads.

The non-triviality resides in the server side interrupt: it can optionally acquire a Power Management wake lock, for a short while, in order to give a chance to applications to process incoming data.

```
    static void
ex_server_xirq_hdl (void* cookie, NkXIrq xirq)
{
    if (ex_dev->wakeup) {
        __pm_wakeup_event(ex_dev->ws, ex_dev->wakeup);
    }
    wake_up_interruptible(&ex_dev->wait);
}
    static void
ex_client_xirq_hdl (void* cookie, NkXIrq xirq)
{
    wake_up_interruptible(&ex_dev->wait);
}
```

5.2.6.5 Managing the Handshake

To establish the connection, both the server and the client vlink end-points must reach the "**ON**" state on both sides (this gives four "**ON**" states), therefore several vlink state transitions have to happen.

These transitions are managed by the threads that open the vbpipe, one in each peer. Opening a vbpipe is a rendezvous between two threads. It is not possible to leave some data in vbpipe like in a mailbox, waiting for the peer to open it and read later.

Code managing this in the `ex_open()` routine is as follows:

```
    if (wait_event_freezable(ex_dev->wait, ex_link_ready(ex_dev))) {
```

with the `ex_link_ready()` routine being the handshake managing routine.

The `wait_event_freezable()` is not a function, but a macro, which invokes the second argument many times, until it returns a Boolean "true" state. This Boolean true state of course means that all four vlink states are **ON**. If a signal is sent to the thread, the macro terminates immediately and returns an error code, typically `-ERESTARTSYS`.

The first invocation happens before attempting to wait, and triggers the state machine, transitioning vlinks from **OFF** to **RESET** state and sending a cross interrupt to the peer:

```
    switch (*my_state) {
        case NK_DEV_VLINK_OFF:
            if (peer_state != NK_DEV_VLINK_ON) {
                ex_ring_reset(ex_dev, server);
                *my_state = NK_DEV_VLINK_RESET;
                ex_sysconf_trigger(ex_dev);
            }
            break;
```

The `ex_sysconf_trigger()` function is a wrapper for an NKDDI call:

```
    static void
ex_sysconf_trigger(ExDev* ex_dev)
{
    nkops.nk_xirq_trigger(NK_XIRQ_SYSCONF, ex_dev->s_link->c_id);
}
```

If the peer still did not open its end of the vbpipe, the thread waits, on the specified `ex_dev->wait` queue. This queue is awakened by the **SYSCONF** handler:

```
    static void
ex_sysconf_hdl(void* cookie, NkXIrq xirq)
{
    ExDev*   ex_dev;
    unsigned i;
    for (i = 0, ex_dev = ex_devs; i < ex_dev_num; i++, ex_dev++) {
        if (ex_dev->enabled) {
            wake_up_interruptible(&ex_dev->wait);
        }
    }
}
```

After each wakeup, the `wait_event_freezable()` macro invokes the `ex_link_ready()` function to see if further waiting is necessary.

When the peer eventually opens the vbpipe, it enters a **RESET** state too, sends us a cross interrupt, which leads us to **ON** state, after which peer enters **ON** as well.

Let's note that we do not care if the peer reboots or is not even booted; the thread waits in `open()` until the peer vlink reaches the appropriate state.

If the peer already opened its end, it updates its state and sends back a cross interrupt, which allows us to enter **ON** state and waits for peer to reach **ON** as well.

Note a corner case might occur if the sender close before the receiver exit from handshake: in that case the receiver will see that the sender state is **OFF while it is in ON state.** In that case the open will succeed letting a chance to the receiver to read the data.

Here is the full code:

```
    static int
ex_handshake (ExDev* ex_dev, _Bool server)
{
    volatile int* my_state;
    int           peer_state;
    if (server) {
        my_state   = &ex_dev->s_link->s_state;
        peer_state =  ex_dev->s_link->c_state;
    } else {
        my_state   = &ex_dev->c_link->c_state;
        peer_state =  ex_dev->c_link->s_state;
    }
    switch (*my_state) {
        case NK_DEV_VLINK_OFF:
            if (peer_state != NK_DEV_VLINK_ON) {
                ex_ring_reset(ex_dev, server);
                *my_state = NK_DEV_VLINK_RESET;
                ex_sysconf_trigger(ex_dev);
            }
            break;
        case NK_DEV_VLINK_RESET:
            if (peer_state != NK_DEV_VLINK_OFF) {
                ex_ring_init(ex_dev, server);
                *my_state = NK_DEV_VLINK_ON;
                ex_sysconf_trigger(ex_dev);
            }
            break;
    }
    return (*my_state  == NK_DEV_VLINK_ON) &&
           (peer_state == NK_DEV_VLINK_ON ||
            peer_state == NK_DEV_VLINK_OFF);
}
    static int
ex_link_ready(ExDev* ex_dev)
{
    int s_ok = ex_handshake(ex_dev, 1);
    int c_ok = ex_handshake(ex_dev, 0);
    return s_ok && c_ok;
}
```

The `ex_ring_reset()` routine performs a cleanup and initialization of the ring variables in **PMEM** which we own. In general, we cannot expect the **PMEM** to contain zeros initially, because PMEM

content survives virtual machine reboots, and contains zeros at the very first VM boot only. So, a reset is required at every **OFF** to **RESET** vlink state transition:

```
    static void
ex_ring_reset (ExDev* ex_dev, _Bool server)
{
    if (server) {
        ex_dev->s_ring->s_idx  = 0;
        ex_dev->s_ring->s_wait = 0;
        ex_dev->s_pos          = 0;
    } else {
        ex_dev->c_ring->c_idx  = 0;
        ex_dev->c_ring->c_wait = 0;
        ex_dev->c_pos          = 0;
    }
}
```

We see in this routine all the control fields of the shared `ExRing` structure in **PMEM**:

```
typedef struct ExRing {
    volatile nku32_f    s_idx;  /* "free running" "server" index */
    volatile nku32_f    s_wait; /* flag: "server" might go to sleep */
    volatile nku32_f    c_idx;  /* "free running" "client" index */
    volatile nku32_f    c_wait; /* flag: "client" might go to sleep */
    nku8_f      ring[MIN_RING_SIZE]; /* circular communication buffer */
} ExRing;
```

The `ex_ring_init()` routine, which is empty in this driver and hence not quoted, is invoked when the peer has already initialized its own variables in `PMEM`, because it is now in the `RESET` state. So we can read his variables and possibly do something with them, like saving a local shadow copy or, like in the "paddle" vdriver, deciding who is supposed to throw the ball first, cf. the "3.6 The Game" section.

The `ex_link_ready()` routine is written to get both `ex_handshake()` invocations performed every time. Indeed, they have side effects, which is the transitioning of vlink states. With C language short-circuit evaluation mechanism for the `&&` operator, if the first function returned false, the second function would not be called. This would delay the second vlink initialization until the first one has completed. This would not be a defect but would not be very efficient.

5.2.7 Write Operation

5.2.7.1 Introduction

The `ex_write()` callback function implements the write operation required by Linux semantics for character devices:

```
    static ssize_t
ex_write (struct file* file, const char __user* buf,
          size_t count, loff_t* ppos)
{
    /* . . . */
```

The function transfers all `count` bytes to the reader side if operating in blocking mode. In non-blocking mode (`O_NONBLOCK` set), it writes as much as possible, and returns the number of written bytes .

It returns the `-EPIPE` error if the reader is not okay. It waits until there is room to write in the circular buffer. When awaken, it checks the peer state and exits if it is not **ON**. If the connection is okay, it writes as many characters as possible and waits again until all `count` bytes have been transferred. It sends a cross interrupt to peer if circular buffer was empty, so peer driver can get more characters from the circular buffer.

5.2.7.2 Mutual Exclusion

First, the code makes sure there is only 1 active writer at a time. This assures that writes will be atomic.

```
    unsigned int minor   = iminor(file->f_path.dentry->d_inode);
    ExDev*       ex_dev  = &ex_devs[minor];
    if (mutex_lock_interruptible(&ex_dev->wlock)) {
        return -EINTR;
    }
```

A second writer will block until the first one is unblocked, even if the vbpipe is currently in `O_NONBLOCK` mode.

5.2.7.3 Atomicity for Small Writes

For non-blocking small writes, which means that `count` is at most equal to the shared memory size (`ex_dev->c_size`), the code ensures atomicity and returns the `-EAGAIN` error code:

```
    if ((file->f_flags & O_NONBLOCK) && (count <= ex_dev->c_size) &&
                (RING_P_ROOM(cring, ex_dev->c_size) < count)) {
        mutex_unlock(&ex_dev->wlock);
        return -EAGAIN;
    }
```

where `RING_P_ROOM()` gives the available byte space for the "producer" or writer:

```
    #define RING_P_ROOM(rng,size)    ((size) - ((rng)->c_idx - (rng)-
>s_idx))
```

The `poll(2)` or `select(2)` system calls then allow to wait for space. Support for them is implemented by the `ex_poll()` callback in the vbpipe driver.

This mechanism makes it possible to avoid handling partial writes in the writer, which otherwise would need to loop until all bytes are transferred.

However, if there is another writing thread, wanting to write a smaller amount of data, which can fit the current spare size, this thread would then send it first.

5.2.7.4 Ring-writing Code Loop

Given the circular nature of the shared memory and the linearity of the passed source buffer, transferring data from user provided buffer to shared memory and then peer may require several copies and waiting for peer to take the data out.

Care must be exercised with signals: if a writer is still in the middle of waiting for space but has already put some data into the ring, it must not return an error code to the caller upon interruption, because the caller would then receive a -1 status and lose track of transferred bytes. It would not be able to resume operations. Similarly, `-EAGAIN` should not be returned in case of non-blocking writes in such circumstances: it is returned at the next `ex_write()` invocation, when there are still no transferred bytes.

Instead, the code must return the number of transferred bytes. The Linux kernel takes care of returning this value to user space and running the pending signal handler in user space.

The main loop looks like this, with `count` being the original buffer size all along:

```
    ssize_t      done;
    size_t       cnt;

    done = 0;
    while ((cnt = (count - done))) {
        /* . . . */
    }
    return done;
```

For non-blocking writes, we exit from the write loop when there is no more room in the circular buffer or if peer side (server vlink side) is no more in **ON** state:

```
        if (file->f_flags & O_NONBLOCK) {
            if (clink->s_state != NK_DEV_VLINK_ON) {
                done = -EPIPE;
                break;
            }
            if (RING_P_ROOM(cring, ex_dev->c_size) == 0) {
                if (done == 0) {
                    done = -EAGAIN;
                }
                break;
            }
        }
```

Note that in case of link-loss, we return an `-EPIPE` error immediately, rather than `done`, the already written byte count. Indeed, we would not be able to resume the transfer (and it is unlikely that the peer has done something with received bytes).

We arrive to the point in code where the writing probably have to wait for space inside buffer. This is a good place to eventually send a *can-receive* cross interrupt to peer, to signal the data we have put into ring previously, if any.

In general, we want to delay sending this interrupt as much as possible, so that peer can retrieve as much data as possible in one go, so we additionally test for no remaining writing space in ring:

```
        if (xirq_needed && RING_P_ROOM(cring, ex_dev->c_size) == 0) {
            xirq_needed = 0;
            nkops.nk_xirq_trigger(ex_dev->c_s_xirq, clink->s_id);
        }
```

The `xirq_needed` variable is set the first time we put something into ring. It is examined before exiting from `ex_write()` and the cross interrupt is sent if still pending.

Now, wait until we can write and exit if something is wrong. We use the same `wait_event_freezable()` macro which we previously used in `ex_open()` and which evaluates the passed condition before and after every sleep until it becomes true:

```
        cring->c_wait = 1;
        if (wait_event_freezable(ex_dev->wait,
                        (RING_P_ROOM(cring, ex_dev->c_size) != 0) ||
                        (clink->s_state != NK_DEV_VLINK_ON))) {
            done = -EINTR;
            cring->c_wait = 0;
```

```
            break;
        }
        cring->c_wait = 0;
```

The condition here is "having some writing space in the ring or link no more OK".

Additionally, we use a variable to signal peer that we are actively waiting and want to receive a cross interrupt on change in ring, as described in "4.5 Optimal Signaling" section. This is different from `ex_open()`, which receives the **SYSCONF** interrupts unconditionally.

The SMP memory barrier, necessary between setting `c_wait` (which is the "`write_waiting`" variable from section "4.6 Almost Final Algorithm") and examining the available room, is hidden inside `wait_event_freezable()`.

The `wait_event_freezable()` macro returns a non-zero value on signal. In this case, we return back to user mode to deliver it[12]. Otherwise, we need to examine which of the 2 conditions has satisfied the wait. If the vlink state is bad, then we abort, otherwise we assume there is space for writing in ring.

```
        if (clink->s_state != NK_DEV_VLINK_ON) {
            done = -EPIPE;
            break;
        }
```

Let's note that if the vlink exits the **ON** state (and the only valid exit state is **OFF**), it cannot return back to **ON** unless we perform the handshake through the `ex_open()` procedure; nothing is performed by the **SYSCONF** handler. Therefore, the **OFF** condition is stable.

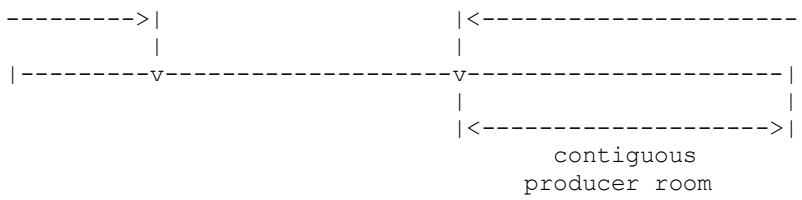
We now have *some* space in the ring, however the exact *continuous* space which can be written by a single copy-in operation from the user space buffer still needs to be computed. Indeed, the `RING_P_ROOM()` macro does not take into account the circularity of the **PMEM** buffer: it only deals with free-running indexes and the total buffer size. Some of the free space might be at the very end of the buffer and the rest at the very beginning. We can only write a chunk size that is the minimum of these 3 sizes:

- `count - done`
- available room
- contiguous room

This is pictured below:

```
          s_idx                c_idx
            |                    |       available
            |                    |     producer room
```

[12] It seems that `done` should not be set to error code if non-zero, otherwise we lose the count of previously transferred bytes.



and gives the following code. Given that `cnt` is the difference between `count` and `done` computed in the `while()` statement for current iteration, we just need to adjust it down two times:

```
room = RING_P_ROOM(cring, ex_dev->c_size);
if (room < cnt) {
    cnt = room;
}
room = RING_P_CROOM(ex_dev);
if (room < cnt) {
    cnt = room;
}
```

The `RING_P_CROOM()` macro is constructed using `c_pos`, the current writing index inside the shared memory. The continuous space goes from this position until the size of the shared memory:

```
#define RING_P_CROOM(ex_dev)    ((ex_dev)->c_size - (ex_dev)->c_pos)
```

The `c_pos` value could be computed from `c_idx`, which is the current free-running "write_index", using the `c_idx % c_size` operation, but the we maintain it separately, including the wrapping back to 0 when it reaches `c_size`.

Once we know how much we can write, we can perform the copy-in from user buffer directly into the ring:

```
if (copy_from_user(&cring->ring[ex_dev->c_pos], buf, cnt)) {
    done = -EFAULT;
    break;
}
```

Then we increment the indexes, to prepare the copying of the next chunk, if any. A memory barrier must be used between the copy-in operation and the `c_idx` "write_index" increment, so that the reader sees the new buffer content before being allowed to access it:

```
buf           += cnt;
mb();
cring->c_idx  += cnt;
done          += cnt;
ex_dev->c_pos += cnt;
if (ex_dev->c_pos == ex_dev->c_size) {
    ex_dev->c_pos = 0;
}
```

It remains to tell our peer that we put new content into buffer, if we just put something into an empty buffer. For this, we need to send it a cross interrupt, but only if the "read_waiting" variable `s_wait` is set. As you remember from section "4.7.1 Metadata Coherence", a memory barrier has to be inserted between modifying the "write_index" and reading "read_waiting". To recognize a situation where the buffer was initially empty, we just compare what is inside (as reported by the `RING_C_ROOM()` macro) with what we just put there (`cnt`).

```
mb();
if (RING_C_ROOM(cring) == cnt &&
```

```
            cring->s_wait == 1) {
            xirq_needed = 1;
        }
```

with:

```
#define RING_C_ROOM(rng)   ((rng)->c_idx - (rng)->s_idx)
```

Following the previously mentioned optimization, we do not actually send a cross interrupt immediately, but instead remember to do it before exiting, or when the buffer becomes full.

The above completes the main loop. Before exiting, we have to send the pending cross interrupt, if any, and to release the `wlock` write mutex.

```
    if (xirq_needed) {
        nkops.nk_xirq_trigger(ex_dev->c_s_xirq, clink->s_id);
    }
    mutex_unlock(&ex_dev->wlock);
    return done;
```

The actual driver code also maintains some statistics, which have been omitted here.

You may have noticed that as far as the sending of the *can-receive* cross interrupt is concerned, the introduction of the delaying through the `xirq_needed` variable introduces a possible loss of sync with the reader: by the time we actually send the xirq, the reader may have already stopped waiting, for example because it has received a signal. Improving this is left as an exercise to the reader.

5.2.8 Read Operation

Implementing the reading for the vbpipe is very similar to implementing writing. A reader can be assimilated to a writer who provides free space instead of data. Therefore, some variables just have to be swapped. However, we have one extra semantic which does not exist on write side.

The `ex_read()` callback handler implements the read operation required by Linux for files.

```
    static ssize_t
ex_read (struct file* file, char __user* buf,
         size_t count, loff_t* ppos)
{
    /* . . . */
```

It attempts to read all `count` bytes from writer side. It can return less than required bytes if writer side is not ok. It waits until there is something to read in the circular buffer. When awaken, it checks the peer's state and exits with less than required bytes if peer is not **ON**. If the communication link is ok, it reads as many characters as possible and waits again until all `count` bytes have been transferred. It sends a cross interrupt to peer if required, so that the peer driver can put more characters into it.

Just like with writing, the routine ensures that only one thread executes the read operation, so that parallel threads reading a long buffer do not steal small chunks from each other:

```
    if (mutex_lock_interruptible(&ex_dev->rlock)) {
        return -EINTR;
    }
```

The main read loop is in charge of transferring all `count` bytes, taking whatever chunks are available in the ring:

```
    done = 0;
    while ((cnt = (count - done))) {
        /* . . . */
    }
    return done;
```

For non-blocking reads, we exit from the read loop when there are no more characters in the circular buffer, returning `-EAGAIN` if we could not read anything and the link is still `ON`, or the count of read bytes, potentially 0, which means end of file:

```
        if ((file->f_flags & O_NONBLOCK) &&
            (RING_C_ROOM(sring) == 0)) {
            if ((done == 0) && (slink->c_state == NK_DEV_VLINK_ON)) {
                done = -EAGAIN;
            }
            break;
        }
```

The read side is different from the write side as it processes the optional "`a`" flag inside the vlink "`info`" configuration field. This flag enables the `WAIT_ONLY_ON_EMPTY` read policy. By default, vbpipe reading waits until the buffer has been fully satisfied, as what happens for example in case of disk reads. When the `WAIT_ONLY_ON_EMPTY` policy is enabled, `read()` returns to user the chars available in the buffer as soon as they are available, which is how network sockets behave by default.

With the default policy, it is not necessary to perform multiple reads when a specific number of bytes is absolutely required, for example to fill a C language structure.

```
        if (!(done && (ex_dev->flags & WAIT_ONLY_ON_EMPTY))) {
```

If this condition is true, we wait until there is something to read and exit if something is wrong.

Like with writes, we differ the sending of *can-transmit* interrupts, until we have taken all the data from the ring. For memory, we send such interrupts when the ring goes from full to no-more-full status.

```
            if (xirq_needed && RING_C_ROOM(sring) == 0) {
                xirq_needed = 0;
                nkops.nk_xirq_trigger(ex_dev->s_c_xirq, slink->c_id);
            }
```

We wait until there are bytes to read and exit in case of link loss. We signal our waiting state to our peer, so that it can wake us up with cross interrupt:

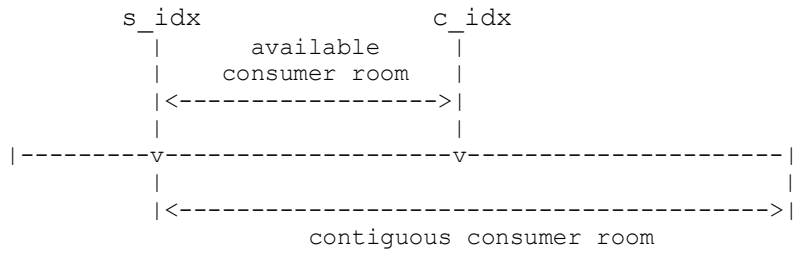
```
            sring->s_wait = 1;
            if (wait_event_freezable(ex_dev->wait,
                                     (RING_C_ROOM(sring) != 0)  ||
                                     (slink->c_state !=
NK_DEV_VLINK_ON))) {
                sring->s_wait = 0;
                done = -EINTR;
                break;
            }
            sring->s_wait = 0;
        }
```

The above exits with `-EINTR` only on signal reception.

Even if vlink peer `c_state` is not `NK_DEV_LINK_ON` anymore, we should continue to read as long as `RING_C_ROOM()` is non-zero. The read operation returns less than the requested `count` bytes and further reads return 0 bytes, which means "end of file". These are `pipe(2)` semantics.

Once we get data, we check how many bytes we can transfer to user space with one linear copy-out operation. Indeed, `RING_C_ROOM()` reported a total amount, without caring about the fact that part of available bytes might be at the end of ring memory, and the remaining ones at the beginning, because of wrapping. We can only take the minimum of `cnt (or count - done)`, available bytes and contiguous bytes.

This is pictured below:



and gives the following code:

```
room = RING_C_ROOM(sring);
if (room < cnt) {
    cnt = room;
}
room = RING_C_CROOM(ex_dev);
if (room < cnt) {
    cnt = room;
}
```

with:

```
#define RING_C_CROOM(ex_dev)     ((ex_dev)->s_size - (ex_dev)->s_pos)
```

which is identical to `RING_P_CROOM`, except for `s` replacing `c`.

We exit if there are no bytes to read. This can happen in two cases: if link `c_state` is not **`ON`** or when the `WAIT_ONLY_ON_EMPTY` policy was specified for device and there are no chars left to read.

```
if (cnt == 0) {
    break;
}
```

We copy to user buffer directly from server ring, and exit if we have a problem, e.g. an invalid buffer address.

```
if (copy_to_user(buf, &sring->ring[ex_dev->s_pos], cnt)) {
    done = -EFAULT;
    break;
}
```

Following the copy, we can increment our `"read_index"`, but must issue a memory barrier first, to make sure peer only sees the incremented index after we read the data, and does not overwrite them with new ones:

```
buf            += cnt;
mb();
```

```
        sring->s_idx  += cnt;
        done          += cnt;
        ex_dev->s_pos += cnt;
        if (ex_dev->s_pos == ex_dev->s_size) {
            ex_dev->s_pos = 0;
        }
```

In parallel, we increment and possibly wrap-around the `s_pos` index used for accessing the ring buffer.

It remains to wake up the writer if it is blocked on a full ring situation. We detect a full ring situation by comparing what we read with current available space for writing. If this is the same value, the ring was full. For peer wake up, we need to send it a cross interrupt, but only if the "write_waiting" variable `c_wait` is set. As you remember from section "4.7.1 Metadata Coherence", a memory barrier has to be inserted between modifying the "read_index" and reading "write_waiting".

```
        mb();
        if (RING_P_ROOM(sring, ex_dev->s_size) == cnt &&
            sring->c_wait == 1) {
            xirq_needed = 1;
        }
```

The above concludes the reading loop. Before returning to caller, we still have to push the pending cross interrupt to peer, if any.

```
    if (xirq_needed) {
        nkops.nk_xirq_trigger(ex_dev->s_c_xirq, slink->c_id);
    }
```

Finally, we unlock the reader and return the read size:

```
    mutex_unlock(&ex_dev->rlock);
    return done;
```

5.2.9 Non-blocking Operations

A real-world program have to wait on several blocking vbpipe-like system objects. A realistic and efficient implementation is rather not dedicate threads to wait on a single object only. This also introduces the issue of easily shutting down threads at process termination time, as each of them have to be tracked and signaled.

Instead, it either opens the vbpipe in non-blocking mode by passing the `O_NONBLOCK` option to `open(2)`, or later set this option using the `fcntl(2)` system call:

```
    int flags = fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

Let's note that the `O_NONBLOCK` flag is an option of the open file and not of the device. Each open file handle can be set differently, however both the `read(2)` and `write(2)` operations will have the same mode. The vbpipe driver retrieves the appropriate value during the I/O operation and adapts its behavior:

```
    if (file->f_flags & O_NONBLOCK) . . .;
```

by returning the `-EAGAIN` error code instead of blocking for data or space in ring.

The user mode process is then supposed to use the `select(2)`, `pselect(2)` or `poll(2)` system calls to wait for the file descriptor to become readable, writable or in some special state.

Given that these calls can operate on several file descriptors at the same time, the kernel avoids sleeping inside a specific driver.

Instead, the Linux kernel calls the `ex_poll()` function of our driver:

```
    static unsigned int
ex_poll(struct file* file, poll_table* wait) . . .
```

It calls it at least twice, once before sleeping and later on wakeups, and each time the function returns a bitmask status indicating readability, writability and other conditions, like vlink loss or "hang-up", HUP in short:

```
    if (ex_dev->s_link->c_state != NK_DEV_VLINK_ON ||
        ex_dev->c_link->s_state != NK_DEV_VLINK_ON) {
        return (POLLERR | POLLHUP);
    }
    sring->s_wait = 1; wmb();
    if (RING_C_ROOM(sring) != 0) {
        sring->s_wait = 0;
        res |= (POLLIN | POLLRDNORM);
    }
```

The difference between these calls is that the first time, Linux passes a non-NULL `poll_table*` parameter, and expects the driver to call back the `poll_wait()` function, passing it the `wait_queue_head_t` object associated with events on the device:

```
    poll_wait(file, &ex_dev->wait, wait);
```

The Linux framework attaches the current "selecting/polling" thread's `wait_queue_t object` to driver's wait queue head, using for example the `add_wait_queue()` call, and at some point calls `schedule()` to wait for wakeup.

Of course, it does not sleep if somewhere during this process one of the "poll" functions returns interesting events.

To avoid race conditions, the polling thread has to attach to the wait queue head before checking for events, so as not to miss wakeups. This is standard Linux kernel condition variable wait procedure. Notably, the kernel sets the thread's `TASK_INTERRUPTIBLE` status before calling `ex_poll()` and the driver's interrupt handler clears it back to `TASK_RUNNING` state during the `wake_up_interruptible()` call on wait queue head, preventing `schedule()` from holding the thread.

An additional difficulty arises from the optimization introduced in section "4.5 Optimal Signaling" about sending *can-transmit* and *can-received* interrupts: the peer only sends us a cross interrupt if our "write_waiting" and "read_waiting" shared variables are set.

Therefore, we must set these variables if we want to eventually be awakened. For efficiency reasons, we should remove them as soon as we obtained the desired state.

The following piece of code sets the `s_wait` and `c_wait` variables *before* testing the ring conditions, as explained in "4.5 Optimal Signaling" section, then tests for data (`RING_C_ROOM`) or for space (`RING_P_ROOM`), and if a given condition is satisfied, immediately removes the corresponding "waiting" variable:

```
    sring->s_wait = 1; wmb();
    if (RING_C_ROOM(sring) != 0) {
        sring->s_wait = 0;
        res |= (POLLIN | POLLRDNORM);
    }
    cring->c_wait = 1; wmb();
    if (RING_P_ROOM(cring, ex_dev->c_size) != 0) {
        cring->c_wait = 0;
        res |= (POLLOUT | POLLWRNORM);
    }
    return res;
```

Therefore, we can have 4 possible outcomes: either both variables are set, or just one of the 2, or none. In a process which multiplexes I/O on several file descriptors, the "none" state is the most frequent one, as usually only 1 in N "selected" file descriptors is signaled at a time and N-1 are not.

Given that the kernel always calls `ex_poll()` on wakeup, to see and report what happened, the waiting variables are cleared, to avoid further useless cross interrupts from peer.

For proper SMP operations, a memory barrier has to be inserted between setting the "waiting" variables and examining the ring indexes through the `RING_C_ROOM` or `RING_P_ROOM` macros. This is explained in section "4.7 Final Algorithm with SMP Support" above.

5.2.10 Ioctl Implementation

The `ioctl(2)` call is implemented by the `ex_ioctl()` callback and currently only allows to see how many bytes are available for reading, as some networking code likes to be able to peek at data without retrieving it. This is the `FIONREAD` operation. Just like with Linux pipes, no error is generated after the write end of the vbpipe has been closed.

```
    static long
ex_ioctl (struct file* file, unsigned int cmd, unsigned long addr)
{    /* . . . */
    switch (cmd) {
    case FIONREAD:
        return put_user(RING_C_ROOM(ex_dev->s_ring), (int*) addr);
    /* . . .*/
}
```

5.2.11 Close Implementation

The `close(2)` call is implemented with the `ex_release()` callback. At every possibly nested close, the device usage `count` counter is decremented and at the last close, it shuts the vlinks down.

Even if `read(2)` and `write(2)` operations can detect that something is wrong with peer driver (it went to `NK_DEV_VLINK_OFF` state), they only exit with a null size or a `-EPIPE` error code in this case, but they do not change the local link state. Applications must perform appropriate actions for errors, then call the `close(2)` function to change local vlink state to `NK_DEV_VLINK_OFF`. After that the application may call `open(2)` to reopen the vbpipe and initiate a new communications session.

```
    static int
ex_release (struct inode* inode, struct file* file)
{
```

```
    if (mutex_lock_interruptible(&ex_dev->olock)) {
        return -EINTR;
    }
    ex_dev->count -= 1;
    if (ex_dev->count == 0) {
        ex_dev_cleanup(ex_dev);
    }
    mutex_unlock(&ex_dev->olock);
    return 0;
}
```

The `ex_dev_cleanup()` function is also used for failed initializations, so it performs a careful cleanup, releasing only already initialized resources. It is called only from `ex_open()` and `ex_release()` functions, and Linux guarantees that there is no other activity on this device yet or anymore. Interrupt handlers have to be detached. The usage count is zeroed; this is for failed initializations, not for final cleanup, when it should be zero already. After resetting our side of both vlinks, we must inform the peer by sending it a **`SYSCONF`** cross interrupt.

```
    static void
ex_dev_cleanup(ExDev* ex_dev)
{
    if (ex_dev->s_s_xid != 0) {
        nkops.nk_xirq_detach(ex_dev->s_s_xid);
    }
    if (ex_dev->c_c_xid != 0) {
        nkops.nk_xirq_detach(ex_dev->c_c_xid);
    }
    ex_dev->count = 0;
    ex_dev->s_link->s_state = NK_DEV_VLINK_OFF;
    ex_dev->c_link->c_state = NK_DEV_VLINK_OFF;
    ex_sysconf_trigger(ex_dev);
}
```

6 Third Example: Using VRPC for Inter VM calls

This chapter describes a virtual driver which lets a client VM get the time from a server VM. This driver relies on a so-called VRPC virtual driver which is delivered along with the Hypervisor. The source code (`vexample3-be.c` and `vexample3-fe.c`) is provided along with this documentation.

6.1 Principles

The mechanisms described in the previous chapters allow low-level programming of virtual device drivers and constructing of arbitrary processing. In many cases though, the interaction between virtual machines remains simple and could be constructed above higher-level services easier to use.

For example, if only synchronous remote calls from a front-end in a VM to a back-end in another VM are required, the VRPC driver could be used as a basis, instead of programming at the vlink level. VRPC stands for Virtual Remote Procedure Calls. The API of the VRPC virtual driver is described in the VRPC(4D) section of the Virtual Device Driver Reference Manual.

This driver is delivered as part of the `vdrivers` component, located in the `src/vdrivers-3.18/drivers/vlx` subdirectory. It is composed of:

- a `vrpc.h` header file
- an `include/vlx/vrpc_common.h` protocol file describing **`PMEM`** content
- a `vrpc.c` source file

Only one remote call can be performed at a time. A single data buffer is prepared by the caller in shared memory, delivered to the server. The server modifies this buffer and returns it to the caller.[13]

The same driver is used for both back-end and front-end purposes. Roles depend on the vlink configuration in the Hypervisor device tree. A given virtual machine can be both a server and a client, depending on the needs of a specific communication protocol.

Users of the VRPC driver do not need to have both the front-end and the back-end parts inside the same driver.

Here is a sample configuration in the device tree:

```
&vm2_vdevs {
    vexample3_be: vexample3@be {     // VM2 vexample3 back-end
        compatible    = "vrpc";      // uses generic vrpc protocol
        server;
        info          = "vexample3"; // service name
   };
};
&vm3_vdevs {
    vexample3@fe {                       // VM3 vexample3 front-end
        peer-phandle = <&vexample3_be>; // peer vlink
        client;
        info          = "vexample3";     // service name
    };
};
```

This establishes a single vlink between VM2 acting as a server (back-end) and VM3 acting as a client (front-end). The vlink is going to be generically named "`vrpc`", but every instance has a different `info` field which indicates the final service, "`vexample3`" in this case, and potentially any other service name, which depends solely on the users of the VRPC driver. This allows to share the VRPC service between several vdrivers.

Virtual drivers using the VRPC service must exist on both the back-end and the front-end sides, and they will find each other thanks to the "`vexample3`" service name.

[13] There is no automatic generation of client-side "stub" and server-side "skeleton" code and no Interface Definition Language (IDL). This could be implemented above VRPC code.

6.2 Interface

VRPC automates:

- scanning of the vlinks database to find those matching a *service name*
- connecting back-end to front-end asynchronously (vlink handshake)
- notifying the client when this connection is established
- matching a reply with a call
- running a server thread in the back-end

The vlink info field for VRPC has the following syntax:

```
info = "name[,[pmem_size][,extra_parameters]]"
```

The `pmem_size` can be *`value`*`K` or *`value`*`M`. The actual size is the maximum of values passed on client and server sides of the vlink definition. A 1 Kb minimal size is enforced, for protocol overhead fields, and `pmem_size` is with overhead included; currently the overhead size is 12 bytes..

The `extra_parameters` are passed to the clients of the VRPC driver and can be anything.

Observation is possible through the `/proc` filesystem on both vlink ends. Here is an example of `/proc/nk/vrpc` in VM2 where the VRPC driver is actively used by our example (`vrtc`):

```
Pr Id OTUCRH Sta Size Calls RxBytes- TxBytes- Info
 3  6 .S...N Off  3f4     0        0        0 vexample3
 3  0 OSUC.I  On  3f4     3      136      136 vrtc
```

and here is the matching `/proc/nk/vrpc` in VM3:

```
Pr Id OTUCRH Sta Size Calls RxBytes- TxBytes- Info
 2  6 .C...N Off  3f4     0        0        0 vexample3
 2  0 OCU.RW  On  3f4     3      136      136 vrtc
```

Legend for columns:

- `Pr` = peer guest id
- `Id` = vlink id
- `O/.` = open/not-open (open API call performed, and no `vrpc_close()` yet)
- `C/S` = client/server (type)
- `U/.` = used/not-used (lookup API call performed, and no `vrpc_release()` yet)
- `C/.` = server call routine defined
- `R/.` = ready/not-ready
- `H: N/D/I/W` = *null*/*direct*/*indirect*/*wakeup* = no handler defined yet, direct delivery from xirq in server, delivery through server thread, client mode
- `Sta` = vlink state (**`OFF`**, **`RESET`**, **`ON`**)
- `Size` = shared buffer size (payload only)

The meaning of these columns is explained as we discuss example drivers below.

6.3 Remote Clock Vdrivers

The two drivers `vexample3-be` and `vexample3-fe` based on VRPC implement a Remote Clock:

- The client (front-end) opens the server to be notified when the time changes on the server, every second in practice. Then, it calls the server to retrieve the time.
- The server (back-end) notifies the client every second about time change and waits for invocations for time retrieval. Time is obtained from the system clock.

Rather than describing all the API calls exported by the VRPC device driver, we present the calls in the order in which they are used. Again, refer to the VRPC(4D) man page for a complete description of the VRPC API.

All the VRPC calls are exported from the `vrpc.c` source file through Linux `EXPORT_SYMBOL()` macros.

6.3.1 Back-end Server Initialization

The server must find all vlinks (VRPC clients) to serve using a VRPC-specific call:

```
#define   VEXAMPLE3_VRPC_NAME   "vexample3"

    static int __init
vexample3_be_init (void)
{
    struct vrpc_t* vrpc = NULL;
    while ((vrpc = vrpc_server_lookup (VEXAMPLE3_VRPC_NAME, vrpc)) != NULL) {
        if (!vexample3_be_create (vrpc)) {
        }
    }
    return 0;
}
module_init (vexample3_be_init);
```

Passing a `NULL` name instead of `VEXAMPLE3_VRPC_NAME` is possible, but selects only nameless vlinks, without `info` vlink field or without name inside.

The `vexample3_be_create` allocates a descriptor associated to the vrpc end-point discovered, storing the received vrpc descriptor, the address of the associated data area (`vrpc_data`) and the size of this area:

```
    static int __init
vexample3_be_create (struct vrpc_t* vrpc)
{
    vexample3_be_t*     vexample3;
    int                 diag;

    vexample3 = (vexample3_be_t*) kzalloc (sizeof (vexample3_be_t), GFP_KERNEL);
    vexample3->vrpc         = vrpc;
    vexample3->vrpc_data    = vrpc_data (vrpc);
    vexample3->vrpc_maxsize = vrpc_maxsize (vrpc);
```

`kzalloc()` is used to have a stable initial zero content and to be able to detect not yet initialized fields during cleanup on error. Of course, the real driver contains error handling on allocation error, omitted here for brevity.

Given that the size of the shared memory is configurable in the Hypervisor device tree, the driver must check if the user-selected size is large enough for operations. Normally, the user does not need to change the default 1KB size for vexample3 to work, so this check should always succeed:

```
    if (vexample3->vrpc_maxsize < sizeof (vexample3_ipc_t)) {
        ETRACE ("Not enough VRPC shared memory (%lu < %lu)\n",
```

```
                (long) vexample3->vrpc_maxsize,
                (long) sizeof (vexample3_ipc_t));
        vexample3_be_destroy (vexample3);
        return -EINVAL;
    }
```

The `vexample3_ipc_t` type is a C union of all the data types which are ever going to reside in shared memory, for both requests and replies. We will discuss them in detail later:

```
typedef union {
    vexample3_req_t      req;
    vexample3_res_t      res;
    vexample3_res_time_t res_time;
} vexample3_ipc_t;
```

The size of this union is the maximum size we are ever going to need inside the shared memory. Code will not actually `use vexample3_ipc_t` variables; having a union variable avoids to manually compute the maximum shared memory size.

A Linux high-resolution timer is allocated to trigger 1-second notifications from back-end to front-end, once the front-end attaches to the back-end:

```
    hrtimer_init (&vexample3->hrtimer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    vexample3->hrtimer.function = vexample3_be_hrtimer_handler;
```

The `hrtimer_init()` call does not set the handler function, so this is done manually afterwards.

The back-end has to allocate a cross interrupt number to be sent to its clients via `nkops.nk_xirq_trigger()` call towards the front-end[14]:

```
    vexample3->cxirq = nkops.nk_pxirq_alloc (vrpc_plink (vexample3->vrpc),
                                             VRPC_PXIRQ_BASE, vrpc_vlink
                                             (vexample3->vrpc)->c_id, 1);
```

In the above code, we see three new VRPC API elements:

- `vrpc_plink (vexample3->vrpc)` is the accessor function that returns the physical address of the `NkDevVlink` vlink corresponding to an opaque `vrpc_t` object. Having the address of the vlink allows to attach extra resources, like cross interrupts here, or possibly shared memory regions.
- `VRPC_PXIRQ_BASE` is the first user-available cross interrupt number for the underlying vlink object. Cross interrupts numbered from `0` till `VRPC_PXIRQ_BASE-1` are reserved for VRPC internal operations.
- `vrpc_vlink (vexample3->vrpc)` returns the virtual address of the `NkDevVlink` vlink corresponding to a `vrpc_t` object.

The above initializations allow to export the service towards the front-end:

[14] The server does not need to attach a handler for the allocated cross interrupt as it only sends it and never receives it.

```
    diag = vrpc_server_open (vrpc, vexample3_be_process_calls, vexample3, 0);
    vexample3->vrpc_open = 1;
```

The `vexample3_be_process_calls()` routine is a handler for incoming calls, `vexample3` is the opaque cookie parameter passed to it.

Invocations from VRPC client can start before `vrpc_server_open()` returns. This call is hence similar to `nkops.nk_xirq_attach()`.

There is no indication in the server-side API that the client has connected at VRPC level. The client must invoke the server to be noticed. There is only one possible client for each VRPC link.

If the handler routine is left as `NULL` the server will fail all requests.

The last parameter of `0` says that invocations should not be handled directly, which means not directly from the cross interrupt handler triggered by the front-end. Instead, the `vrpc_server_open()` routine allocates a server thread which calls our `vexample3_be_process_calls()` handler. We need a thread context because we are going to invoke Linux OS services which must not be invoked from interrupt handlers.

Here is the handler signature:

```
    static vrpc_size_t
vexample3_be_process_calls (void* cookie, vrpc_size_t size)
{ /* . . . */
}
```

Once all the initializations are done, we store our new `vexample3_be_t` service descriptor on a single-linked list. This list is used during driver unloading and possibly in other circumstances as well, like for example to maintain a `/proc/nk/vexample3-be` virtual file, allowing to observe the open connections:

```
    vexample3->next = vexample3_bes;
    vexample3_bes   = vexample3;
    return 0;
```

The above concludes a single vlink initialization process.

Below is the full definition of the data structure initialized above:

```
typedef struct vexample3_be_t {
        /* Variables identical to peer ones */
    struct vrpc_t*      vrpc;
    void*               vrpc_data;
    vrpc_size_t         vrpc_maxsize;
    NkXIrq              cxirq;
    struct vexample3_be_t*      next;
    _Bool               vrpc_open;
        /* Variables different from peer ones */
    struct hrtimer      hrtimer;
} vexample3_be_t;

static vexample3_be_t*  vexample3_bes;
```

6.3.2 Back-end Server Termination

Each `vexample3_be_t` object needs cleanup at driver unloading time and this cleanup must occur in opposite order from initialization. Objects might be only partially initialized in case of failure, so the cleanup function takes all possibilities into account.

```
    static void
vexample3_be_destroy (vexample3_be_t* vexample3)
{
    vexample3_be_t** link = &vexample3_bes;
    while (*link != vexample3 && *link) {
        link = &(*link)->next;
    }
    if (*link) {
        *link = vexample3->next;
    }
```

The above list-removal code considers the possibility of the `vexample3` object not yet present in the list. Last object removal clears the `vexample3_bes` list pointer.

There is no way to know from `vrpc_t` whether it is already open or not, so a state variable is used, which was set in `vexample3_init()` earlier:

```
    if (vexample3->vrpc_open) {
        vrpc_close (vexample3->vrpc);
    }
```

This will terminate the server thread and set the vlink to **OFF** state.

Cancelling the `hrtimer_t` timer waits until current handler execution, if any, ends:

```
    if (vexample3->hrtimer.function) {
        hrtimer_cancel (&vexample3->hrtimer);
    }
```

Releasing the `vrpc_t` object allows the `vrpc_server_lookup()` function to find it again, at next invocation. The connection becomes no more "used":

```
    vrpc_release (vexample3->vrpc);
    kfree (vexample3);
```

The global release process is driven by simple loop. Given that `vexample3_bes` is updated by `vexample3_be_destroy()`, the loop ends up naturally on a `NULL` pointer.

```
    static void __exit
vexample3_be_exit (void)
{
    while (vexample3_bes) {
        vexample3_be_destroy (vexample3_bes);
    }
}
```

6.3.3 Back-end Server: Processing Remote Invocations

A single handler receives all invocations from client. In our case, this handler is called in "indirect" mode, meaning in a dedicated thread context, as requested during `vrpc_server_open()`, so we are free to do anything inside.

```
    static vrpc_size_t
vexample3_be_process_calls (void* cookie, vrpc_size_t size)
{
    vexample3_be_t*     vexample3 = (vexample3_be_t*) cookie;
    vexample3_req_t*    req  = (vexample3_req_t*) vexample3->vrpc_data;
    vexample3_res_t*    res  = (vexample3_res_t*) vexample3->vrpc_data;
    vrpc_size_t         ret;
```

The `vrpc_size_t` parameter and return value are the current amount of data in the `vrpc_data` **PMEM** shared memory. Different remote call invocations can pass and return different sizes.

Given that the same buffer is used for requests and for replies, care must be exercised to not overwrite arguments with results when they are still necessary. To avoid such problems, code can potentially put separate data structures inside `vrpc_data`. We do not use this approach here.

Here are the definitions of arguments and of results structures:

```
typedef struct {
    nku32_f vcmd;      /* vexample3_cmd_t */
} vexample3_req_t;

typedef enum {
    VEXAMPLE3_CMD_OPEN,
    VEXAMPLE3_CMD_CLOSE,
    VEXAMPLE3_CMD_READ_TIME
} vexample3_cmd_t;

typedef struct {
    nku32_f res; /* result */
} vexample3_res_t;
```

Every request starts with a `vcmd` field to indicate the operation being requested and allowing to understand other fields. This allows to implement different calls. Indeed, the VRPC protocol only allows clients to pass shared memory content and its size.

As most, if not all, requests can fail internally in the server, all replies start with a common `vexample3_res_t` structure containing a result field (`res`).

Given that all requests must contain at least enough data for the base type, we can perform a common quick validity check:

```
    if (size < sizeof (vexample3_req_t)) {
        ETRACE ("VRPC request size error (size %d req %d)\n",
                size, (int) sizeof (vexample3_req_t));
        return 0;
    }
```

Returning a null size from handler usually means an error, though it is up to client to verify. Normally, a client expects a specific reply size for every request, at least as large as `vexample3_res_t`.

We leave a console trace in DEBUG mode about currently performed request. We cannot trust the `req->vcmd` field for being correct, so we must not directly use it to index the `vexample3_be_cmd_name` array of command names.

```
    DTRACE ("VM%d vcmd=%u(%s) \n", vrpc_peer_id (vexample3->vrpc),
            req->vcmd, vexample3_be_cmd_name [req->vcmd % VEXAMPLE3_CMD_MAX]);
```

The `vrpc_peer_id()` function can be used in both the server and the client. There is only one client and one server for a given VRPC link.

The server accepts 3 requests: open, close and read time. The "open" request starts a 1 second relative timer:

```
    switch (req->vcmd) {
    case VEXAMPLE3_CMD_OPEN:
        TRACE ("Open from VM%d\n", vrpc_peer_id (vexample3->vrpc));
```

```
        hrtimer_start (&vexample3->hrtimer, ktime_set(1, 0),
                       HRTIMER_MODE_REL);
        res->res = 0;
        ret = sizeof *res;
        break;
```

The `res` field of the VRPC answer is cleared, to indicate success. The return value of the handler is set to the size of `vexample3_res_t` to indicate size answer. We could check the `hrtimer_start()` return value and whether the server back-end is still closed, and issue error in such cases.

The `res->res = 0` assignment overwrites the `req->vcmd` field, but this is not an issue.

The "close" request cancels the timer, possibly waiting for current handler to terminate:

```
    case VEXAMPLE3_CMD_CLOSE:
        TRACE ("Close from VM%d\n", vrpc_peer_id (vexample3->vrpc));
        hrtimer_cancel (&vexample3->hrtimer);
        res->res = 0;
        ret = sizeof *res;
        break;
```

We could check whether the server back-end is opened.

A different transaction is necessary for reading the current time value. Though the request structure is the same, an additional field is required for the reply:

```
typedef struct {
    vexample3_res_t     common;
    nku64_f             time;
} vexample3_res_time_t;
```

We must recast the VRPC buffer to the `vexample3_res_time_t` pointer for storing the reply.

The `getnstimeofday()` call provides current time in VM, which we can return to caller. This is the time set with `date(1)`, so during test execution you can test how changes in server propagate to client.

```
    case VEXAMPLE3_CMD_READ_TIME: {
        vexample3_res_time_t* res2 = vexample3->vrpc_data;
        struct timespec       wall_time;
        getnstimeofday (&wall_time);
        res2->time = wall_time.tv_sec;
        res2->common.res = 0;
        ret = sizeof *res2;
        break;
    }
```

The handler should return the result's size, and `0` in case of some fundamental errors, like an invalid request:

```
    default:
        ret = 0;
        break;
    }
    DTRACE ("VM%d ret=%d res->res=%d\n",
            vrpc_peer_id (vexample3->vrpc), ret, res->res);
    return ret;
```

Fields `res->res` and `res2->common.res` are aliases for the same memory location in the shared buffer.

6.3.4 Back-end Server: Sending Notifications to Client

The vexample3 VRPC client requests the current time only after the server has notified it about a change through a cross interrupt issued from the `hrtimer` callback handler:

```
    static enum hrtimer_restart
vexample3_be_hrtimer_handler (struct hrtimer* hrtimer)
{
    vexample3_be_t* vexample3 = container_of (hrtimer, vexample3_be_t, hrtimer);
    nkops.nk_xirq_trigger (vexample3->cxirq,
                           vrpc_vlink (vexample3->vrpc)->c_id);
    hrtimer_forward_now (hrtimer, ktime_set (1, 0));
    return HRTIMER_RESTART;
}
```

After sending the interrupt, the handler reprograms itself to run again in exactly 1 second. The `hrtimer_forward_now()` function computes the new deadline taking into account the theoretical time at which the current handler has started, so there is no cumulative drift. Forgetting this function and returning `HRTIMER_RESTART` would result in an infinite busy loop and kernel panic.

6.3.5 Front-end Client Initialization

A driver implementing a VRPC client needs to call `vrpc_client_lookup()` to enumerate all the vlinks for a specific protocol.

```
    static int __init
vexample3_fe_init (void)
{
    struct vrpc_t* vrpc = NULL;
    unsigned       used = 0;
    while ((vrpc = vrpc_client_lookup (VEXAMPLE3_VRPC_NAME, vrpc)) != NULL) {
        if (!vexample3_fe_create (vrpc)) {
        }
    }
    return 0;
}
module_init (vexample3_fe_init);
```

As for `vrpc_server_lookup()`, passing a `NULL` protocol name selects only nameless vlinks.

The `vexample3_fe_create()` vlink front-end creation function is similar to the back-end one. It first allocates a local descriptor. This descriptor has the same VRPC control fields as the back-end version:

```
    static int __init
vexample3_fe_create (struct vrpc_t* vrpc)
{
    vexample3_fe_t*     vexample3;
    int                 diag;
    vexample3 = (vexample3_fe_t*) kzalloc (sizeof (vexample3_fe_t), GFP_KERNEL);
    vexample3->vrpc         = vrpc;
    vexample3->vrpc_data    = vrpc_data (vrpc);
    vexample3->vrpc_maxsize = vrpc_maxsize (vrpc);
```

Correct minimal buffer size is verified as well:

```
    if (vexample3->vrpc_maxsize < sizeof (vexample3_ipc_t)) {
            /* . . . */
        return -EINVAL;
    }
```

Opening a VRPC client is different from opening a vrpc server. The VRPC code synchronizes clients with servers, which might not yet be running. The `vrpc_client_open()` routine either waits until server is present if no callback routine is provided or returns immediately and ensures the provided "ready" callback is invoked when the server happens to be ready. This callback is performed from a thread and a cookie is passed to it:

```
    diag = vrpc_client_open (vrpc, vexample3_fe_ready, vexample3);
    vexample3->vrpc_open = 1;
```

In our `vexample3_fe_create()` routine, we use a callback "ready" function[15]which opens the connection to the server at the logical level. As indicated previously, VRPC servers are not notified when their client connects, so an explicit initialization is necessary. The `vrpc_client_open` is the first VRPC call to server., It is discussed in details later. For now, we accept that it can either succeed or fail. If the open succeeds, we remember this fact, so as to perform the close operation later, as cleanup:

```
    static void
vexample3_fe_ready (void* cookie)
{
    vexample3_fe_t* vexample3 = (vexample3_fe_t*) cookie;
    if (!vexample3_fe_op_open (vexample3)) {
        vexample3->server_open = 1;
    }
}
```

For tracking resources to free at driver unloading, we maintain a list of VRPCs:

```
    vexample3->next = vexample3_fes;
    vexample3_fes   = vexample3;
```

Each time the front-end driver receives a cross interrupt from server it asks the server for the current date and time. For this, we need to operate at thread level, so we allocate a `struct work`. Given that this `work` is going to be signaled from interrupts, it must be initialized before interrupts happen:

```
    INIT_WORK (&vexample3->work, vexample3_fe_work);
    vexample3->work_inited = 1;
    vexample3->cxirq = nkops.nk_pxirq_alloc (vrpc_plink (vexample3->vrpc),
                                             VRPC_PXIRQ_BASE, vrpc_vlink
                                             (vexample3->vrpc)->c_id, 1);
    vexample3->cxid = nkops.nk_xirq_attach (vexample3->cxirq,
```

[15] When reconnecting after a failed call, a synchronous open is used. This is illustrated further in this document.

```
                                                vexample3_fe_cxirq_handler,
                                                vexample3);
```

Unlike in back-end, we not only allocate a cross interrupt, but also attach the handler. This ends the initialization process.

```
    return 0;
```

Below is the full definition of the data structure initialized above:

```
typedef struct vexample3_fe_t {
        /* Variables identical to peer ones */
    struct vrpc_t*      vrpc;
    void*               vrpc_data;
    vrpc_size_t         vrpc_maxsize;
    NkXIrq              cxirq;
    struct vexample3_fe_t*      next;
    _Bool               vrpc_open;
        /* Variables different from peer ones */
    NkXIrqId            cxid;
    struct work_struct  work;
    _Bool               work_inited;
    _Bool               server_open;
} vexample3_fe_t;

static vexample3_fe_t*  vexample3_fes;
```

6.3.6 Front-end Client Termination

All local descriptors are linked behind the `vexample3_fes` list, so we can just iterate:

```
    static void __exit
vexample3_fe_exit (void)
{
    while (vexample3_fes) {
        vexample3_fe_destroy (vexample3_fes);
    }
}
module_exit (vexample3_fe_exit);
```

where `vexample3_fe_destroy()` is a cleanup function releasing acquired resources, and doing so in opposite order of allocating, making sure to stop asynchronous activities before destroying objects and waiting until asynchronous code (interrupts and works) stop executing:

```
    static void
vexample3_fe_destroy (vexample3_fe_t* vexample3)
{
    vexample3_fe_t** link = &vexample3_fes;
    if (vexample3->cxid) {
        nkops.nk_xirq_detach (vexample3->cxid);
    }
    if (vexample3->work_inited) {
        cancel_work_sync (&vexample3->work);
    }
```

This list removal code works even if `vexample3` is not yet on the list:

```
    while (*link != vexample3 && *link) {
        link = &(*link)->next;
    }
```

```
    if (*link) {
        *link = vexample3->next;
    }
    if (vexample3->server_open) {
        vexample3_fe_op_close (vexample3);
    }
    if (vexample3->vrpc_open) {
        vrpc_close (vexample3->vrpc);
    }
```

Releasing the `vrpc_t` object allows the `vrpc_client_lookup()` function to find it again, at next invocation. The connection becomes no more "used".

```
    vrpc_release (vexample3->vrpc);
    kfree (vexample3);
}
```

6.3.7 Front-end Client: Issuing Remote Invocations

We have seen earlier the "server ready" callback invoke an open routine for the server. Here is the implementation of this routine. The bulk of the work is inside a common handler named `vexample3_fe_call()` and the specific "open" routine only processes variable, call-dependent parts of the work:

```
    static int
vexample3_fe_op_open (vexample3_fe_t* vexample3)
{
    vexample3_res_t* res = vexample3->vrpc_data;
    vrpc_size_t      size;
    size = vexample3_fe_call (vexample3, VEXAMPLE3_CMD_OPEN,
                              sizeof (vexample3_req_t));
    if (size != sizeof (vexample3_res_t) || res->res) {
        return -EFAULT;
    }
    return 0;
}
```

We pass the *request* size `sizeof (vexample3_req_t)` to `vexample3_fe_call`, and check returned `size` value against the expected *result* size, which is `sizeof (vexample3_res_t)`, a different type. Both data types occupy the same memory buffer at `vexample3->vrpc_data`.

Error checking requires verifying whether remote processing has succeeded (`res->res` being zero). The result is transcoded to a single kernel-like return value.

The "close" routine is even simpler, as we do not care about the outcome and perform best-effort work only:

```
    static void
vexample3_fe_op_close (vexample3_fe_t* vexample3)
{
    vexample3_fe_call (vexample3, VEXAMPLE3_CMD_CLOSE,
                       sizeof (vexample3_req_t)));
}
```

In the `vexample3_fe_call()` function, the `vcmd` field needs to be set in shared memory before the `vrpc_call` occurs. All remote requests have this field in the same place in shared memory, so we can delegate setting it to a common function. Argument `size0` is the size of arguments and

needs to be passed to the `vrpc_call()` API by pointer. On return, the size argument will be updated with the value returned by server. The `vrpc_call()` call does all the invocation synchronization work.

```
    static vrpc_size_t
vexample3_fe_call (vexample3_fe_t* vexample3, const vexample3_cmd_t vcmd,
                   const vrpc_size_t size0)
{
    struct vrpc_t*      vrpc = vexample3->vrpc;
    vexample3_req_t*    req  = vexample3->vrpc_data;
    vrpc_size_t         size;
    if (!vexample3->vrpc_open) return 0;
    for (;;) {
        req->vcmd = vcmd;
        size      = size0;
        if (!vrpc_call (vrpc, &size)) {
            break;
        }
        ETRACE ("Lost backend. Closing and reopening.\n");
        vrpc_close (vrpc);
        if (vrpc_client_open (vrpc, 0, 0)) {
            vexample3->vrpc_open = 0;
            size = 0;
            break;
        }
        TRACE ("Re-established backend link.\n");
    }
    return size;
}
```

6.3.8 Error Handling During Remote Invocations

The `vrpc_call()` call can fail and return a negative error value for reasons unrelated to the actual performed remote operation:

- server has been unloaded from memory
- server VM rebooted
- client user thread issuing the VRPC call was interrupted while waiting for the server to reply
- unloading of client code is attempted
- …

Our server thread being a kernel one, immune to signals, we will not get interrupted, but other cases are possible.

The vexample3 caller tries to recover the call failure by closing and re-opening the client connection. A `NULL` "ready" function pointer is passed to `vrpc_client_open()`, which means that the calling thread is going to be blocked until success, or some higher-order failure, like front-end driver unloading from memory.

If re-opening fails, the driver enters an error mode by clearing `vexample3->vrpc_open`, which is tested before every invocation. All future remote invocations will fail immediately. The driver must be unloaded and re-loaded for recovery.

More ambitious error recovery policies are possible and are left as an exercise to the reader.

6.3.9 Front-end Client: Main Operating Code

We have seen that the "open" operation on the server has started sending a periodic cross interrupt towards the front-end, to signal time change. Processing this interrupt requires a remote VM invocation. Hence it is systematically deferred to a Linux "work":

```
    static void
vexample3_fe_cxirq_handler (void* cookie, NkXIrq xirq)
{
    vexample3_fe_t* vexample3 = (vexample3_fe_t*) cookie;
    queue_work (system_wq, &vexample3->work);
}
```

The `work` routine was defined through `INIT_WORK()` in `vexample3_fe_create()`:

```
    static void
vexample3_fe_work (struct work_struct* work)
{
    vexample3_fe_t* vexample3 = container_of (work, vexample3_fe_t, work);
    nku64_f rtime;
    if (!vexample3_fe_op_read_time (vexample3, &rtime)) {
        struct rtc_time tm;
        rtc_time_to_tm (rtime, &tm);    /* void */
        TRACE ("VM%d time is %llu or %d/%02d/%04d %d:%02d:%02d\n",
               vrpc_peer_id (vexample3->vrpc), (long long) rtime,
               tm.tm_mday, 1 + tm.tm_mon, 1900 + tm.tm_year,
               tm.tm_hour, tm.tm_min, tm.tm_sec);
    }
}
```

It performs a remote invocation and in case of success displays the remote time on the console, both as seconds since UNIX Epoch and as split time in European 24-hour style.

The implementation of the remote call for retrieving time is shown hereafter:

```
    static int
vexample3_fe_op_read_time (vexample3_fe_t* vexample3, nku64_f* rtime)
{
    vexample3_res_time_t* res = vexample3->vrpc_data;
    vrpc_size_t           size;
    size = vexample3_fe_call (vexample3, VEXAMPLE3_CMD_READ_TIME,
                              sizeof (vexample3_req_t));
    if (size != sizeof (*res) || res->common.res) {
        return -EFAULT;
    }
    *rtime = res->time;
    return 0;
}
```

It uses a more complicated version of the result data structure and reads the "`time`" field from it. It also expects the remote invocation to return the size of this specific `vexample3_res_time_t` structure, rather than the size of the generic `vexample3_res_t` one.

6.4 Other VRPC Calls

Some VRPC functionality is not used in the vexample3 drivers.

Function `const char* vrpc_info(vrpc_t*)` returns parameters following the VRPC service name in the vlink `info` field, located after first comma, if any. This allows to customize the operation of each VRPC vlink.

The `VRPC_PXIRQ_BASE` constant can be used to designate the first available PMEM shared memory segment id for a VRPC vlink with the `nkops.nk_pmem_alloc()` call. Such a shared memory segment can be used across several remote calls. Alternatively, some memory could be permanently reserved in the main shared segment, after the zone used for call arguments and results.

Function `int vrpc_call_non_interruptible (vrpc_t*, vrpc_size_t*)` can be used to implement functions which should not fail at protocol level, for example a close, or which are invoked while a mutex is being held.

It is possible to be asynchronously notified about the death of a client or a server by using this setup call:

```
typedef void (*vrpc_off_t) (NkOsId peer);
void vrpc_off_notifier (struct vrpc_t* vrpc, vrpc_off_t off, int use_worker);
```

Given that the `vrpc_off_t` handler only receives an `NkOsId` and not a `vrpc_t*`, it is necessary to examine all vlinks leading to `peer` to find those which are no more in **ON** state.

6.5 VRPC Driver

VRPC shared memory starts with a control block:

```
nku32_f req; /* request counter */
nku32_f ack; /* acknowledge counter */
nku32_f size;      /* size of RPC in/out data */
```

which accounts for the overhead reducing the available payload size. This control block is immediately followed by user data:

```
nku32_f data[];    /* RPC data */
```

The `req` counter field is set by the client end, while the `ack` counter field is set by the server end. Together, they compensate for spurious interrupts:

- make sure that a new request has really been received from the front-end
- make sure that back-end has really finished processing current request

The `size` field allows to determine the current request size, is the return value from handler function, is used for error control and supports a *null request* for waking up the peer in case of shutdown.

The global **SYSCONF** cross interrupt is used for vlink handshake. It also wakes up the caller in case of server death.

There are two per-peer xirq handlers used for signaling: xirq 0 in vlink for server and xirq 1 for client.

Internal states:

- *aborted* = the VRPC module is being unloaded, the module exit routine `_vrpc_exit()` was called, everybody must end.
- *open* = An open call was performed on server or client side.

On Linux, the `vrpc_server_lookup()` and `vrpc_client_lookup()` routines will initialize the VRPC driver if not yet done. This solves an uncertainty about module initialization order and the `module_init()` macro.

There is a limitation in client side: the thread used to report the "ready" state is not waited for before exiting.

7 Virtual Driver Writing Guidelines

This chapter provides a set of rules and recommendations to help you write virtual drivers without too much burden.

7.1 Correctness and Robustness

- Use memory barriers when sharing multiple variables across VMs, as on an SMP system, modifications of variables could appear in the other VM in a different order than in originating VM.
- Do not use spin-locks for variables shared across VMs.
- You can use atomic ops on variables in PMEM, there is NKDDI library support for that:

```
nkops.nk_atomic_clear()
nkops.nk_clear_and_test()
nkops.nk_atomic_set()
nkops.nk_atomic_sub()
nkops.nk_sub_and_test()
nkops.nk_atomic_add()
```

- An atomic operation is only needed when a shared variable is modified by both sides. A good practice is to avoid atomic operations if possible, at least at driver run time, because they are expensive.
- Do not trust **PMEM** content, take snapshots of control variables and validate before use. Otherwise one bad or malicious peer virtual driver may be able to crash you, notably by modifying a variable between the moment when you validated its value and the moment when you used it for something else.
- For the same reason, never put into **PMEM** your own private variables. Have separate data structures for shared **PMEM** content and for private variables.
- Avoid leaving in **PMEM** data when virtual drivers are both removed. This can be a problem with some virtual drivers, like vbpipe, which lose data if peer closes.
- Any **PMEM** location should be written from only one VM, except if you use NKDDI atomic operations (see above) or explicitly ping-pong the ownership of message-like memory blocks between VMs (the Virtual Message Queues driver does this). Decide in which vlink state it is set. Avoid race conditions and avoid curing them by looking at memory content.
- Memory allocated from **PMEM** requires `nk_mem_map()`, and `nk_mem_unmap()` at cleanup time, while memory allocated from **PDEV** needs only `nk_ptov()` and nothing at cleanup time. This also applies to `nk_vlink_lookup()` memory, located in **PDEV**.
- If a virtual driver is loaded into memory, managed vlinks must be in RESET or ON state, otherwise the Hypervisor will consider the virtual driver is absent and will not forward it sysconf interrupts from peers.

- Avoid using `NkOsId` to designate a peer in a multi-client back-end. This prevents from having several front-ends in a single VM.

7.2 Proper Linux Semantics and Integration

- Enable Linux lock debugging during development.
- Performing a remote call from some context can be tricky because Linux does not accept such situation: e.g. freezing when a mutex is taken. In such a case, use a non-freezing waiting primitive.
- If you need to create a character device with just one minor, consider using the `miscdevice` API to share a single major.
- `kfree()` does take NULL pointers, just like the C library `free()` call.
- Do not take plain spinlocks from interrupt handlers. Plain spinlocks do not mask interrupts, which can still run and cause recursive locking.

7.3 Initialization and Termination

- Proper initialization order:
  - allocate resources
  - create threads
  - attach cross interrupt handlers
  - attach `/proc/nk` support
- If you plan to fully initialize each vlink separately, be aware that cross interrupt attachment can immediately invoke handlers which might not yet be prepared to deal with all objects.
- Do not forget to release resources on termination.
- On driver exit, cleanup resources in opposite order from initialization.
- The cleanup routine on module exit can be the same as the code for cleaning a failed initialization.
- During blind cleanup, if the initialized state of an object cannot be easily deduced from its content, maintain a separate Boolean to indicate an "initialized" situation of each such object.
- Do not panic/`BUG()` the whole system without reason.

7.4 VM Interoperation

- Select proper level of para-virtualization, typically some stable Linux API which does not evolve much between versions and stays powerful.
- Make a protocol which is Linux version independent:
  - Do not assume enumerated values stay the same between releases, need to convert both ways, dealing with possible mismatches
  - The same applies to layout of structures. Need to convert to portable versions in **`PMEM`**. Such structures have their names prefixed with `Nk`… sometimes in header files, even though they are not part of Hypervisor ("nanokernel") API
- Stay compiler independent: do not put enumeration variables into shared memory, as they can have different sizes. A workaround is to add a "max" field to force 32 bit or 64 bit size,

but then some compilers and code coverage tools will complain that max value is not used/tested e.g. in `switch()` statements.

- In **PMEM**, align each structure field on its size, and the whole structure on its largest member type. This will assure interoperability between 32-bit and 64-bit modes.
- Stay 32-bit/64-bit independent: use explicit sizes for fields in shared structures, instead of e.g. `long`, which is usually defined as 32 bit wide in 32-bit mode and as 64 bit wide in 64-bit mode.

7.5 Observability

- Prepare `/proc/nk/xxx` for observation. Warning: only 4 KB of buffer data are supported by standard Linux API `struct seq_file`.
- `proc_mkdir()` of an existing `/proc` directory will generate a console error trace. Existence can be checked before creation, either using a private tracking variable of your own, if you create the directory as host for several objects or using the `kern_path()` Linux kernel API.
- Creating the same `/proc` file twice will create two files with same name.

7.6 Performance

- Implementation might require intelligence to reduce overhead:
  - pre-fetching and local caching of results obtained from peer driver
  - prefetching might be based on "Markov-chain" probability
  - write behind: defer invoking peer driver while possible
  - synchronously confirm the writability of every object you plan to defer writing to, to avoid late errors
  - only write behind till next read
  - combine multiple-write and multiple-read requests into one

7.7 General Structure

- Try to re-use existing communication-oriented bricks, like vpipe, vbpipe, vrpc, vmq, vipc, vlink-lib, vumem, vrpq, vdriver-lib.
- Try to re-use vbpipe code if you need a similar reliable mechanism between VMs.
- Porting a driver to another Linux version is not just making sure it compiles and runs, but also possibly supports new features and semantics of the new kernel.
- For better plug and play support, consider the driver establishing a `struct platform_driver` and the generic kernel code declaring a `struct platform_device`. This declaration can be done automatically with a device tree as well, and the driver can then define a `platform_device.driver.of_match_table` table.
- When creating support for a non-Linux operating system, consider getting inspiration from drivers for non-Linux systems, for example RTOS-es. Many of the Linux vdrivers exist in simplified form for RTOS-es.
- Current policy for Linux modules is to use version `#ifdefs` to maintain version compatibility for a single source file.

- Avoid global variables. Linux virtual drivers seldom need them. Many functions can be passed opaque "cookie" parameters, which can store a pointer to all driver variable which would otherwise be global. Several kernel data structures contain custom fields with the same role.
- Console is a slow device and a bottleneck, so making a virtual driver print error traces on bad usage at runtime from a user-mode process allows it to DOS the whole system and to flush out maybe more important traces from the history.
- Linux has 3 different virtual driver APIs:
  - one when a virtual driver is linked with kernel. Any non-static symbol is accessible, and the driver becomes GPL licensed.
  - one when it is a GPL module. Only `EXPORT_SYMBOL*()` symbols are accessible.
  - one when it is a non-GPL module. Only `EXPORT_SYMBOL()` symbols are accessible, which is a smaller subset of exported symbols.

  Ask yourself if your driver needs to be a module and what licensing you accept.
- Linux kernel threads have maybe 16 KB of stack only. Do not put large objects there.
- Considering flagging function arguments with `__attribute__((nonnull (1)))` and functions with `__must_check`.

7.8 Common Errors

- The `nkops.nk_prop_get()` and `nkops.nk_prop_set()` calls need to be repeated as long as they return the `NK_PROP_BUSY` error. This applies to writable properties. Look into their man pages for more details about access from interrupt handlers.

8 QNX driver specific issues

- QNX runs virtual drivers as killable processes, so vlink cleanup must be performed manually. Drivers should leave their vlinks as **OFF** on exit and reset them on startup.
- QNX runs certain virtual driver threads in ARM system mode, with certain calls not available.

9 Zero copy data transfer mechanism

Usually, virtual drivers exchange payload data using the **PMEM** shared memory, but this method requires an extra copy to be done from an OS buffer to the shared buffer located in the **PMEM** memory. While this is often acceptable for some virtual drivers where the amount of payload data is relatively small, and/or the communication throughput is not critical, an extra copy might significantly impact the performance of I/O intensive virtual drivers (like virtual storage).

In order to avoid an extra copy to/from the **PMEM** memory for payload data and therefore increase throughput of I/O intensive drivers (and reduce the CPU overhead induced by such an extra copy), the Hypervisor provides virtual drivers with a zero-copy data transfer mechanism described in this section.

9.1 Importing Virtual Machine memory

The zero-copy data transfer mechanism is based on the physical memory importing feature described in this section.

The Hypervisor provides a physical memory sharing mechanism between VMs by mapping physical memory assigned to one VM (exporter) to the intermediate physical address space of another VM (importer). This makes it possible to directly access physical pages of the exporter VM in the importer VM. It is important to underline that this sharing mechanism works only when both exporter and importer VMs use identical second stage MMU mapping, in other words, when the IPA is equal to the PA. The above requirement avoids any possible conflicts between the exporter and importer physical pages in the importer intermediate address space.

The Hypervisor identifies an imported VM memory node by its compatible property set to ***vl,vm-memory***. A virtual platform can include multiple VM imported memory nodes, but each node should declare a separate VM exporter ID.

A ***vl,vm-id*** mandatory property specifies the exporter VM ID. It is prohibited to be a self-importer, in other words, the exported VM ID must not match the virtual platform owner VM ID.

A ***vl,mapped*** optional property specifies whether all exporter memory is statically mapped to the importer intermediate address space, in other words, whether all exporter memory is always accessible for the CPU and DMA accesses in the importer VM. When this property is omitted, the Hypervisor only provisions for the page tables which would be needed to dynamically create mappings in the second stage MMU for the exporter memory at run time using the hypercalls described in the following section.

Note that only the exporter VM is allowed to create such a dynamic mapping in the importer intermediate address space. Thus, with the dynamic mapping mechanism, it is up to the exporter VM to manage accessibility to its own memory from each importer VM. For each particular dynamic mapping created in the importer intermediate address space, the exporter is able to select the type of allowed access: CPU, DMA or both.

The static mapping does not provide any protection of the exporter memory from the importer VM and therefore it should be used with caution. The static mapping is typically used for prototyping purposes to simplify the memory sharing mechanism or when the importer VM is trusted. In the latter case, the static mapping avoids the second stage MMU management overhead at run time.

When the exporter VM is restarted, the Hypervisor automatically invalidates in the second stage MMU all dynamically created mappings of the exporter memory in all importer VMs. The mappings invalidation takes a part in the VM starting sequence in order to ensure that all importer VMs have already completed any outstanding accesses to/from the exporter memory during the VM stopping phase.

```
/{
    vm-memory@3 {  // import & statically map VM3 memory
        compatible = "vl,vm-memory";
        vl,vm-id = <3>;
        vl,mapped;
    };
};
```

It is a good practice to declare the imported physical memory as reserved in the Linux guest device tree. This makes it possible to create the page descriptors as well as kernel virtual mappings for the imported physical pages. The example below shows a reserved memory node created in the guest device tree for the VM3 imported memory.

```
/{
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <1>;
        ranges;
        vm-memory@3 {   // reserve VM3 imported memory
            compatible = "vl,vm-memory";
            reg = <0 0 0>;
            vl,vm-id = <3>;
        };
    };
};
```

The mandatory ***reg*** property is automatically populated by the Hypervisor with the VM3 physical memory layout. The guest device tree node uses the same 'compatible' and ***vl,vm-id*** properties as the virtual platform. This makes it possible to use a unique node for both memory importing and reservation when the virtual platform is integrated (merged) to the unique guest device tree.

```
/{
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <1>;
        ranges;
        vm-memory@3 {   // reserve VM3 imported memory
            compatible = "vl,vm-memory";
            reg = <0 0 0>;
            vl,vm-id = <3>;
            vl,mapped;
        };
    };
};
```

9.2 Dynamic memory granting

In order to be able to use the dynamic memory granting, the caller VM memory has to be imported by the destination VM using a ***vl,vm-memory*** virtual platform node described in the previous section. The memory importing declaration in the virtual platform enables the dynamic memory granting between the exporter and importer VMs. In addition, such a declaration ensures that the Hypervisor has provisioned the page table resources needed for creation of dynamic mappings in the importer VM intermediate address space. It is important to underline that the memory importing requires an identical IPA mapping in both exporter and importer VMs.

Note that only the exporter VM is allowed to dynamically grant access to its physical memory. Thus, it is up to the exporter VM to manage accessibility to its own memory from each importer VM using the `hyp_call_vm_mem_grant` and `hyp_call_vm_mem_deny` hypercalls.

```
int
hyp_call_vm_mem_grant (NkOsId    vmid,
   nku32_f   flags,
```

```
   nku32_f   count,
   NkMemMap* rgns);
int
hyp_call_vm_mem_deny (NkOsId    vmid,
  nku32_f   flags,
  nku32_f   count,
  NkMemMap* rgns);
```

The `hyp_call_vm_mem_grant` hypercall allows to the exporter VM to enable access to a portion of its physical memory from the importer VM specified by the `vmid` argument.

The `flags` argument selects the memory access permissions in terms of direction (read/write) as well as in terms of master access type (CPU/DMA). In particular, the exporter is able to select the type of allowed master access: CPU, DMA or both. This argument is a bitwise combination of the following flags:

- `HYP_VM_MEM_GRANT_FLAGS_READ` allows read access
- `HYP_VM_MEM_GRANT_FLAGS_WRITE` allows write access
- `HYP_VM_MEM_GRANT_FLAGS_CPU` allows CPU access
- `HYP_VM_MEM_GRANT_FLAGS_DMA` allows DMA

The `count` argument defines the number of region descriptors provided by the `rgns` array. Each `NkMemMap` element of the **`rgns`** array defines a contiguous memory region which should be page aligned.

The `hyp_call_vm_mem_deny` hyper call allows to revoke a previously granted access to a portion of the caller physical memory from the VM designated by the `vmid` argument. Similar to the memory granting, in order to be able to use the dynamic memory revoking, the caller VM memory has to be imported by the destination VM using a ***vl,vm-memory*** virtual platform node described in the previous section.

The `flags` argument selects the memory access revoking in terms of master access type (CPU/DMA). In particular, the exporter is able to select the type of revoked master access: CPU, DMA or both.

- `HYP_VM_MEM_GRANT_FLAGS_CPU` denies CPU access
- `HYP_VM_MEM_GRANT_FLAGS_DMA` denies DMA

The `count` argument defines the number of region descriptors provided by the `rgns` array. Each `NkMemMap` element of the `rgns` array defines a contiguous memory region which should be page aligned.

Note that it is not an error to deny access to a memory which has never been granted.

The example code shown below provides two functions `vm_mem_rgn_grant` and `vm_mem_rgn_deny` which grants and denies CPU and DMA access to a given contiguous memory region from a given VM in read/write mode.

```
//
// Grant CPU and DMA access to a given region from a given VM in
read/write mode.
//
    int
```

```
vm_mem_rgn_grant (NkOsId vmid, phys_addr_t addr, size_t size)
{
    NkMemMap rgn;
    unsigned int flags = HYP_VM_MEM_GRANT_FLAGS_READ |
                HYP_VM_MEM_GRANT_FLAGS_WRITE |
                HYP_VM_MEM_GRANT_FLAGS_CPU |
                HYP_VM_MEM_GRANT_FLAGS_DMA;
    rgn.addr = addr;
    rgn.size = size;
    return hyp_call_vm_mem_grant(vmid, flags, 1, &rgn);
}

//
// Deny CPU and DMA access to a given region from a given VM.
//
    int
vm_mem_rgn_deny (NkOsId vmid, phys_addr_t addr, size_t size)
{
    NkMemMap rgn;
    unsigned int flags = HYP_VM_MEM_GRANT_FLAGS_CPU |
                HYP_VM_MEM_GRANT_FLAGS_DMA;
    rgn.addr = addr;
    rgn.size = size;
    return hyp_call_vm_mem_deny(vmid, flags, 1, &rgn);
```

9.3 Granted memory verification and locking

In order to ensure the memory isolation, the memory importer should verify that the granted memory buffer is valid, in other words, that this buffer is really located in the exporter memory space. Without such a checking, a malicious (or buggy) front-end driver can ask back-end driver to perform an I/O operation to a buffer located outside of its own physical memory and this could potentially compromise another VM.

Hypervisor provides a `hyp_call_vm_mem_verify` hyper call which verifies the granted memory validity by returning 0 when given memory ranges belong to a given VM and -1 otherwise.

```
   int
hyp_call_vm_mem_verify (NkOsId    vmid,
                        nku32_f   count,
                        NkMemMap* rgns);
```

Using this hyper call the memory importer back-end driver is able to ensure that the granted memory buffer communicated by the exporter front-end driver is valid.

It should be understood that hyper call checks the memory assignment only and it doesn't check the validity of the second stage translations for the imported memory. As a consequence, a memory access to the imported buffer can potentially cause a page fault or DMA failure. A possible scenario is a malicious (or buggy) front-end driver which doesn't invoke the `hyp_call_vm_mem_grant` hyper call prior to submission of the I/O buffer to the back-end driver.

Another possible scenario is a malicious (or baggy) front-end driver which revokes access to the granted memory by invoking the `hyp_call_vm_mem_deny` hyper call before the I/O operation is compete. It might happen that the back-end driver is not able to recover from such an access fault.

In order to ensure a fault free access to the imported memory in the back-end driver, hypervisor provides two extra hyper calls `hyp_call_vm_mem_verify_and_lock` and `hyp_call_vm_mem_unlock`.

```
    int
hyp_call_vm_mem_verify_and_lock (NkOsId    vmid,
                                 nku32_f   flags,
                                 nku32_f   count,
                                 NkMemMap* rgns);
    void
hyp_call_vm_mem_unlock (nku32_f   count,
                        NkMemMap* rgns);
```

The former hyper call verifies that given memory regions belong to a given VM and, in addition, the second stage translations map these regions with given access rights (defined by the flags argument). When the check is positive, the hyper call locks the translations in order to ensure their validity until a subsequent `hyp_call_vm_mem_unlock` hyper call. Attempt to invalidate locked translations by invoking a `hyp_call_vm_mem_deny` hyper call will fail. Nested locking is not supported, in other words, an attempt to lock already locked memory will fail.

The latter hyper call unlocks translations of given regions in the second stage MMU or/and System MMU which were previously locked by the `hyp_call_vm_mem_verify_and_lock` hyper call.

```
// safe copy from imported buffer to local memory
    int
do_safe_copy_from_vm(NkOsId vmid, phys_addr_t addr, size_t size)
{
    int res;
    NkMemMap rgn;
    unsigned int flags = HYP_VM_MEM_GRANT_FLAGS_READ |
                         HYP_VM_MEM_GRANT_FLAGS_CPU;
    rgn.addr = addr;
    rgn.size = size;
        // verify and lock src buffer
    res = hyp_call_vm_mem_verify_and_lock(vmid, flags, 1, &rgn);
    if (!res) { // src buffer valid ?
            // copy src buffer from [addr, addr+size) to dst buffer
        …
            // unlock src buffer
        hyp_call_vm_mem_unlock(1, &rgn);
    }
    return res; // success
}
```

9.4 Critical vlink handshake

When a virtual driver uses a zero copy data transfer mechanism, a race condition can happen at exporter VM reboot/restart time. It is possible that the virtual driver running in the importer VM continue to perform some outstanding I/O transfers directly to the exporter memory buffer during the exporter guest OS booting time. In order to avoid such a race condition, Hypervisor provides a synchronization mechanism based on the critical vlink.

A vlink end-point can be declared as critical by adding a ***critical*** property to the vlink node. Hypervisor synchronizes the VM booting process with the handshake on each critical vlink. In particular, at VM booting time, Hypervisor checks the state of all critical peer end-points and suspends the VM booting process if at least one of them is **ON**. It is assuming that the virtual driver running on a critical vlink first performs some clean-up actions prior to updating its vlink state to **OFF** or **RESET**. This ensures that all critical virtual drivers running in other VMs are not functional at guest OS booting time.

This is extremely important for a virtual driver which performs a direct I/O transfer to a memory buffer located in another VM (and not within a special **PMEM** region). Such a driver may cause a memory corruption when it is still functional during the guest OS booting time because the buffer used by the driver can be located in memory re-used by the booting guest OS.

Typically, the direct I/O transfer is performed by a back-end driver to the front-end VM memory. Thus, the ***critical*** property is usually added to the client vlink device node. In such a way the VM running the front-end driver is synchronized by Hypervisor with the back-end driver state at VM rebooting time.

The VM restart/reboot synchronization goes through the following steps:

- Hypervisor stops VM, puts all vlinks in **OFF** state and notifies all peer VMs about vlink state transition.
- For each critical vlink, Hypervisor checks the state of remote end-point. The VM restart process is suspended until all critical vlinks are **OFF** or **RESET**.
- Receiving a Hypervisor notification, remote virtual driver performs a clean-up and makes transition the end-point state to **OFF** or **RESET**. As a part of the handshake protocol, the driver sends a notification to Hypervisor signaling the state transition.
- Receiving driver notification, Hypervisor rechecks the state of all critical remote end-points and resumes the VM restart process when all these end-points are in **OFF** or **RESET** state, in other words, when clean-up is completed in all critical virtual drivers.

9.5 Sharing a specific DRAM region

Sometime virtual drivers running in different Guest OSes need to share a DRAM region located at a specific address in the Physical Address (PA) space.

The setup may be achieved in two steps:

1 Map the DRAM region in the Intermediate Physical Address (IPA) space of each VM by specifying a "vl,io-memory" compatible node in the Virtual Platform Device Tree of each VM whose Guest OS needs access to the DRAM region. For example, if the below node is defined in the Virtual Platform Device Tree of a VM:

```
vl,io-memory@ABCDE000 {
    compatible = "vl,io-memory";
    reg = <0x0 0xABCDE000 0x2000>;
};
```

Then the PA region [0xABCD_E000..0xABCD_FFFF] will be mapped at the address 0xABCD_E000 in the VM's IPA space.

Please refer in reference manual to the Virtual Platform I/O Memory Node - "vl,io-memory"' section of the Hypervisor Reference Manual for further details.

2 At the virtual driver initialization time map the IPA region in the Guest OS kernel virtual address space as a regular physical memory region using the standard Guest OS kernel APIs such as ioremap_cache() in Linux, e.g.:

```
ioremap_cache(0xABCDE000 0x2000);
```

Note, that the Hypervisor configuration must ensure that the DRAM region shared between VMs is not allocated to a VM or to Hypervisor it-self as a regular DRAM.
This can be achieved using one of the following alternative methods:

3 Specify the DRAM region as an invalid region using the Hypervisor Device Tree "&invalid_region" node:

```
&invalid_region {
    reg = <0x0 0xABCDE000 0x2000>;
};
```

Please refer to the 'Configuration/Hypervisor Device Tree bindings/Physical Memory Allocation Constraints - "/vlm/memory/" node' section of the Hypervisor Reference Manual for further details.

4 Do not include the DRAM in the available DRAM regions described by the "memory" device of the boot Device Tree, e.g.:

```
memory {
    device_type = "memory";
    reg = <0x0 0x80000000 0x2BCDE000>, <0x0 0xABCE0000 0x54320000>;
};
```

Please refer to the standard Linux Device Tree bindings for further details.