User level PV driver user guide

Version	0.9
Date	<09/21/2018>
Status	<Draft>
Document ID	<Doc.Id.>
Owner	<Nicolas Lavocat>
Approved	<Date Name>

Table of Contents

1 Introduction 3
1.1 Purpose 3
1.2 Glossary 3

2 Generalities about PV drivers 3

3 Some examples 4
3.1 VRPQ 4
3.1.1 Brief architecture considerations 4
3.1.2 How to use VRPQ 7
3.2 VUMEM 15
3.2.1 Brief architecture considerations 15
3.2.2 Concepts/usage 16
3.2.3 Example of usage 17
3.3 VBPIPE 22
3.3.1 Brief architecture considerations 22
3.3.2 How to use vbpipe 23

1 Introduction

1.1 Purpose

The goal of this document is to explain how to write a user level Para Virtualized (PV) driver.

Firstly, general information about user level PV driver will be provided. Then, some inter-domain communication at user-land level will be described.

1.2 Glossary

Term	Definition
PV	Para Virtualized
BE	Back End
FE	Front End

2 Generalities about PV drivers

PV drivers permit to have access to resources related to physical devices, that are usually managed by the hypervisor or other VMs, from user space.

They provide the same type of interface than usual Linux modules (for instance through /dev).

3 Some examples

3.1 VRPQ

3.1.1 Brief architecture considerations

Let’s firstly briefly describe briefly VRPQ usage. For more information, please have a look to the related HLD.

The VRPQ service is a transport layer that provides user applications with remote procedure posts and calls. It has been firstly dedicated and optimized for vOpenGL but can be used by other drivers such as vOMX The following figure describe the usual usage of this:

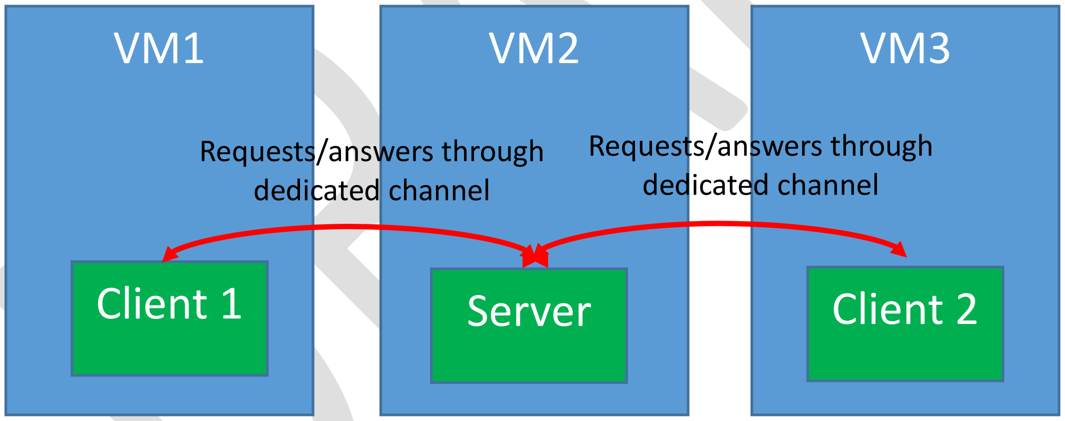
VM1
VM2
VM3
Requests/answers through dedicated channel
Requests/answers through dedicated channel
Client 1
Server
Client 2

Figure 1 VRPQ data transfer

From an architecture point of view, a client needs the following items in order to be able to exchange data with VRPQ:

-a session (usually on per communicating process). A new cession will imply the creation of a new process by the VOpenGL ES daemon on the backend side.

-a channel (on per communicating thread), permitting to have a dedicated communication path between client threads and server instances.

-a port, permitting to multiplex PMEM usage.

Usually, the following sequence is followed in order to use VQRP:

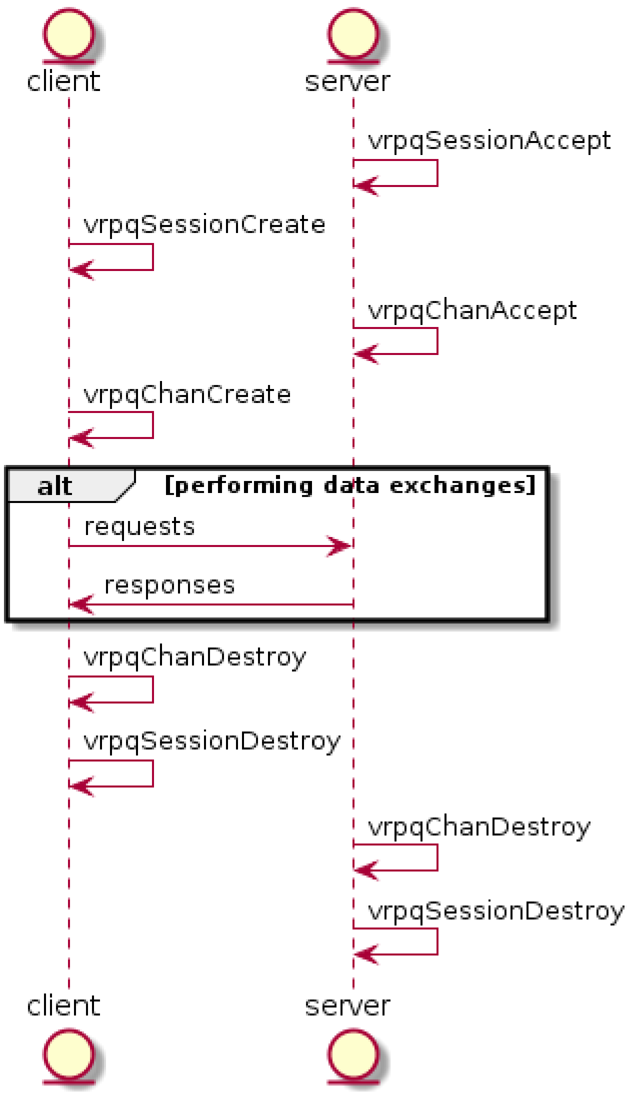

Figure 2 sequence diagram of VRPQ

It can be also interesting to know how requests/ answers are issued between clients and servers:

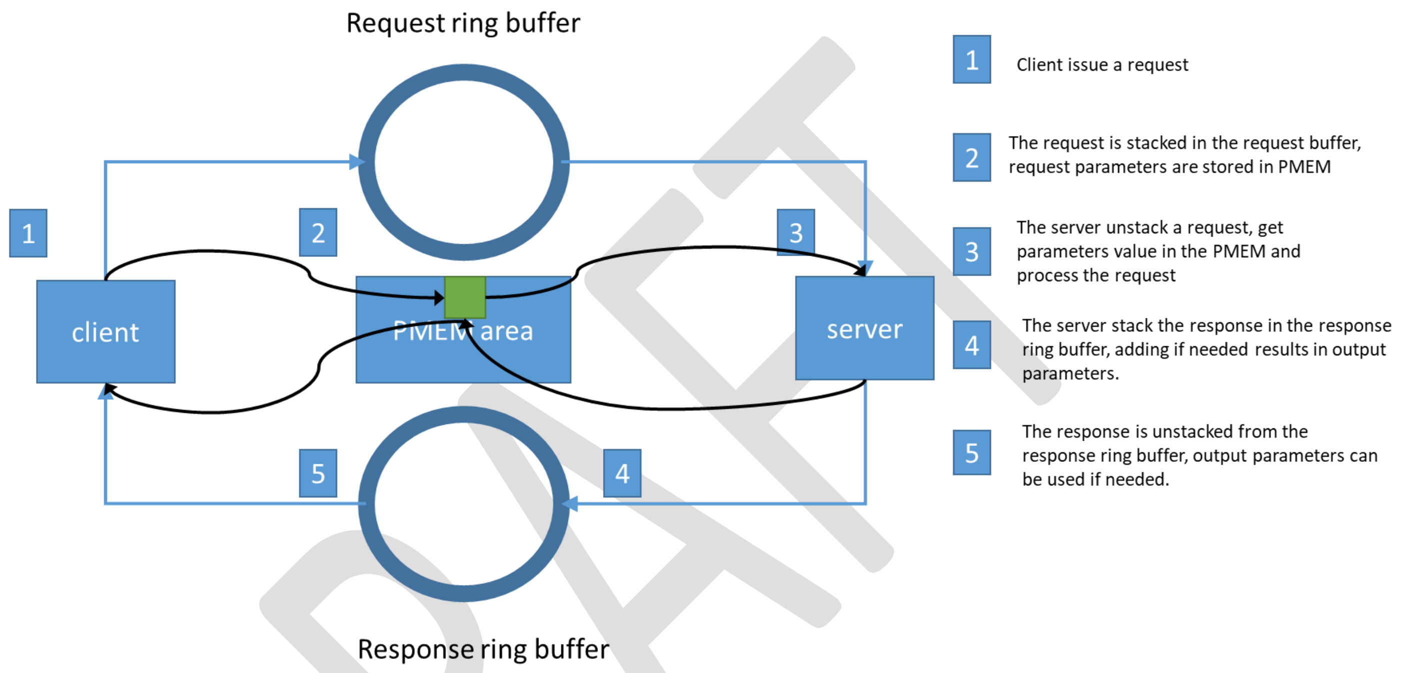
Request ring buffer
1
2
3
client
PMEM area
server
5
4
Response ring buffer
1 Client issue a request
2 The request is stacked in the request buffer, request parameters are stored in PMEM
3 The server unstack a request, get parameters value in the PMEM and process the request
4 The server stack the response in the response ring buffer, adding if needed results in output parameters.
5 The response is unstacked from the response ring buffer, output parameters can be used if needed.

Figure 3 request /response and parameter buffering management

Let's show how to prepare a VM in order to use VRPQ.

3.1.2 How to use VRPQ

3.1.2.1 configuration tuning

First of all, it is necessary to tune the VM configuration.

To do this, a node compatible with `vogl` has to be added in the VM having access to the graphic resources.

The following properties will have to be defined, with a value fulfilling your needs:

- `Vrpq-reqs`: maximum number of requests that the "request rings" will be able to manage
- `Vrpq-pmem`: the size of PMEM allocated for parameters sharing between client and server.

Please find below an example of dts section that specify a `vogl` compatible node, with a request ring buffer having a capacity of 128.000 requests and a PMEM chuck having a size of 2M dedicated to request parameters exchange between clients and servers.

```
&vm2_vdevs {
    vogl_be: vogl@be {                 // vOpenGL back-end for VM3

        compatible = "vogl";           //

      info       = "vumem-pdev=128K", // vUMEM PDEV region size

                     "vrpq-reqs=128K,  //max number of request in the
ring buffer

                  ",vrpq-pmem=2M"; // vRPQ PMEM region size

      server;                     // server end point

   };

};

&vm3_vdevs {

    vogl@fe {                     // vOpenGL front-end

      peer-phandle = <&vogl_be>;  // peer vLINK

      client;                     // client end point
    };
```

Please notice the following points about this dts fragment:

`&vm2_vdev, &vm3_vdev` are references to “node labels” defined elsewhere in the Hypervisor device tree. This allows to attach `vogl` definitions to existing Virtual Machine configuration and avoids #ifdefs

in a single file.

As you can see, 2 parts are necessary in order to declare a `vogl` section: the server part and the client part. Each of them has to be defined per-VM sub-trees of the hypervisor device tree.

In order to match the client to the server, the “node label” mechanism is used again. In the above example, `vogl_be` is the label of the server and VM3 then references this label using peer-phandle property.

Conventionally, the peer-phandle side should me marked as client, and the node-labeled side as server.

If you are using the sadk board, the previous code can be appended to the src/nkernel/ /bsp/vdt/true/vogl.dtsi file which defines the vogl configuration. A good way also is to add only the server definition in the file vogl.dtsi, and client definitions in the file src/nkernel-<vendor> /bsp/board/exynos8/sadk/vdt-vcar.dts file, which defines the device tree for the SADK board hypervisor configuration.

Remark 1: names `vogl@be` and `bogl@fe` have been chosen for the sake of readability, but are absolutely not used during the dts parsing. This is the `compatibility` property that permits to identify that these nodes are related to `vogl` during the parsing.

Remark 2:

A question should come in your mind: how am I supposed to tune these parameters? A way to do this is to set first of all an arbitrary value, and check the number or requests stacked/ the PMEM usage over the time. A tool named vrpqstat can help you in this task, it will be described later.

Let’s have a look to a described code example. We will alternate client and server code sections.

For a better readability, only little pieces of code are described, without checking status return. This is obviously something to avoid in a real life code!

3.1.2.2 Described example

The next section shows an example of simple usage of VRPQ.

The following operations are performed:

- Init of communication

- sending synchronously data from the client to the server
- process a request on server side
- closing communication

Init of Communication

The first step for a client is to initiate the communication with the server, creating a session then a channel:

```
  diag = vrpqSessionCreate(“/dev/vrpq-clt-vogl0”, VRPQ_TEST_PORT_BASE,

                           NULL, 0, &session);
```

/dev/vrpq-clt-vogl0 is a node created by the VRPQ driver in clients. If this node does not exist,

it probably means that no VRPQ client has not been defined for this VM at the configuration step.

Also, "info" and "infoSize" (with values NULL and 0 here) parameters are not used here. It could be, permitting to provide data to the server at the session creation.

At this moment, the server is waiting for session creation with the following call:

```
diag = vrpqSessionAccept(“/dev/vrpq-srv-vogl0”, VRPQ_TEST_PORT_BASE,

                             NULL, NULL, &session);
```

As you can see, the function is completely symmetric to the previous one.

/dev/vrpq-srv-vogl0 is a node created by the VRPQ driver on the server side. Since a server VM can have to deal with multiple client VM, on server node is created per client VM. For instance, if there are 2 client VMs, the following nodes could be found on the server VM:

- “/dev/vrpq-srv-vogl0”
- “/dev/vrpq-srv-vogl1”

Then, this is the turn of the channel creation on the client side:

```
diag = vrpqChanCreate(session, NULL, 0, &chan);
```

Symmetrically, on the server side:

```
 diag = vrpqChanAccept(session, NULL, NULL, &chan);
```

The last init step on the server side is to map in the current process address space the shared memory that is used during a client/server session.

`shm_base` will contain the base address of the shared memory.

```
 diag = vrpqShmMap(session, &shm_base);
```

Init is finished, the server now is waiting for data:

```
for (;;) {

        diag = vrpqReceive(chan, reqs, 8, &avail);

… /*code to be executed when data are coming*/
```

}

Here the server will get at maximum a batch of 8 requests during one reception, `avail` will contain the actual number of received requests and `reqs` the client requests.

The init phase is finished, the client is ready to emit requests and the server to receive them. Let's

See a simple example: sending an asynchronous request.

Sending asynchronously data from the client to the server

Asynchronous requests are the simplest to perform, since they do not expect any feedback from the server.

All requests follow the same pattern:

- Memory allocation of parameters to be sent
- Serialization of the parameters
- Sending of the request

Let's suppose the goal is to transmit a `int` value to the server. The previous steps would be done in the following way:

<u>Memory allocation:</u>

```
in = vrpqParamPostAlloc(chan,
            4, // int size
            0, // no parameter provided by reference
            0);// no reference parameters
```

This API permits to provide information about parameters if there are references (please have a look to the API documentation or to the HLF for further details).

`In` will point to the allocated area.

<u>Serialization of the parameters:</u>

A set of macros is provided for parameter serialization. For a `int`, it will be done in the following way:

```
RPQ_PARAM_WRITE(in, 0, nku32_f, sent_value);
```

Where:

- `In`: the pointer got from the memory allocation described behind.
- `0`: the offset used before writing in the `in` memory (useful if it is necessary to write multiple data in one buffer)
- `nku32_f`: the type of the data being written
- `sent_value`: the value to be written in the in memory

<u>Send the request</u>

The VRPQ library offers the possibility of optimizing the number of data exchanges/Xirq between VM for asynchronous requests, letting the user decides if he wants to pack/buffer its requests before sending them or not.

For this, a request sending is made of two operations: posting and notifying.

- The API vrpqPost permits to just post a request
- The API vrpqNotify permits to notify the server that requests are ready
- The APi vrpqPostAndNotify permits to post the request and notify the server in the same time.

Let’s use the last API for our example, since we will send only once a request.

Also let’s speak about IPC theory:

Usually, each request is identified by procedure ID. This ID must be known by the client and the server. Thus, when the server receives a request, he is able to know exactly what represent the different parameters and how to compute them.

the macro VRPQ_PROC_ID permits to provide the procedure id of the current request, building it from the group id and function id of the request.

This said, please find below a way to send an asynchronous request:

```
  diag = vrpqPostAndNotify(chan, VRPQ_PROC_ID(2,2), in);
```

<u>If you want to dig more:</u>

there are 3 types of transactions in the libvrpq:

- asynchronous requests
- synchronous requests
- callbacks

the first one has been described.

The second one permits to transmit a request from the client, expecting an answer from the server.

The last one permits to send requests from a server to a client. It can be synchronous or asynchronous.

Please have a look to the routines vrpqCallbackHandle (client side) and vrpqCallbackPerform (server side) for further details.

Let’s see how to process a request on the server side.

Process a request on server side

Independently of the nature of the request (synchronous/asynchronous), the way of receiving requests on the server side is the same, like shown in the section Init of Communication :

```
for (;;) {

        diag = vrpqReceive(chan, reqs, 8, &avail);

… /*code to be executed when data are coming*/

}
```

Here the server will get at maximum a batch of 8 requests during one reception, `avail` will contain the actual number of received requests and `reqs` the client requests.

Please note that the call of the function is in a loop, permitting of course to wait for other requests.

After a receive, the server iterates on all requests for instance like this:

```
    for (i = 0; i < (unsigned int) avail; i++, count++) {
```

The next steps are the followings:

- access to the parameters of the request
- identify the request
- decode parameters
- read them
- perform the appropriate action depending in the request and its parameters values.
- Once finished wait for another request

Access to the parameter is done thanks to the following APIs:

```
in  = vrpqParamInGet(req,  shm_base); //Get a pointer to the input
parameters buffer within the client/server shared memory.

out = vrpqParamOutGet(req, shm_base); //Get a pointer to the output
parameters buffer within the client/server shared memory.
```

Then, the request is identified, extracting its group id and function id from its proc id:

```
req     = reqs[i];

proc_id = req->procId;

func_id = VRPQ_PROC_ID_FUNC_GET(proc_id);

grp_id  = VRPQ_PROC_ID_GRP_GET(proc_id);
```

Decoding of parameters can be done thanks to some macro provided by the `libvrpq`:

```
data = VRPQ_PARAM_READ(in, 0, nku32_f);
```

data contains the value sent by the client.

Let’s see the last part: closing communication

Closing communication

The closing of the communication is symmetric to the init.

It can be done by the client or the server. In both case, it is necessary to close firstly the used Channel, then the session.

Whatever the “side”, the following routines have to be called:

```
diag = vrpqChanDestroy(chan);

diag = vrpqSessionDestroy(session);
```

Of course, return status have to be checked every time by the client and the server, in order to detect a potential disconnection.

3.2 VUMEM

VUMEM (for Virtual User MEMory Buffers) is a generic module, providing user-land components with the management of memory buffers. Those memory buffers are intended to be used by hardware devices which are controlled by user-land components.

3.2.1 Brief architecture considerations

In the context of VUMEM, we will call:

- client VM: a VM needing to access to graphic resources, but not having access directly to them
- server VM: a VM having access to graphic resources (HW)

Roughly, VUMEM is composed of four parts:

- VUMEM front end library: used by applications requiring to manage graphic buffer, in a client VM
- VUMEM front end driver:
    - get requests from VUMEM FE library (typically through a node named /dev/vumem-xxx)/transmit the response to the request
    - transmit these request to the VUMEM back end driver of the server VM through the hypervisor
- VUMEM back end driver: symmetric to VUMEM front end driver on server side
- VUMEM back end library: symmetric to VUMEM front end library on server side, used by the graphic resource allocator

The Figure 4 below illustrate an example of interactions between these different parts.

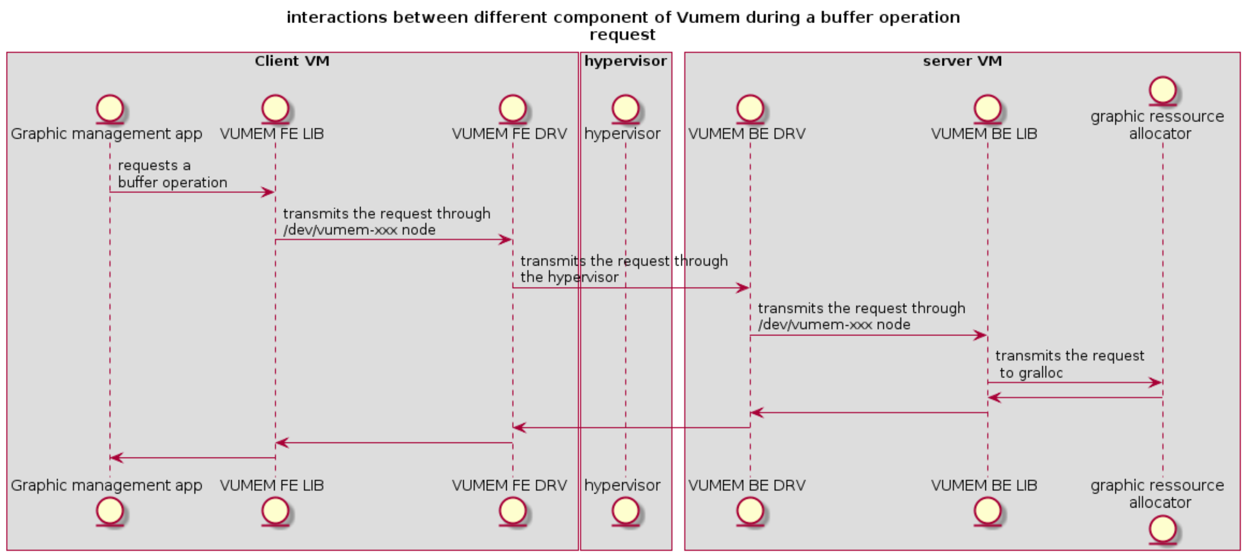
interactions between different component of Vumem during a buffer operation request
Client VM
hypervisor
server VM
Graphic management app
VUMEM FE LIB
VUMEM FE DRV
hypervisor
VUMEM BE DRV
VUMEM BE LIB
graphic ressource allocator
requests a buffer operation
transmits the request through /dev/vumem-xxx node
transmits the request through the hypervisor
transmits the request through /dev/vumem-xxx node
transmits the request to gralloc

Figure 4interactions between different component of VUMEM during a buffer operation request

Thus, all operations are made synchronously.

3.2.2 Concepts/usage

3.2.2.1 What is a session?

A session is a limited period of time during which a client is connected to a server in order to communicate.

Client/server messages are exchanged through an implicit (to the session) and unique communication channel.

Resources created in the context of a session exist as long as the session is alive. Such resources are freed upon session termination.

Such session handle is local to the user entity.

There are 2 kinds of sessions: Master sessions and standard sessions.

If a Master session is allocated, its allocator will be the only one able to perform buffer allocations.

If there are only standard sessions, then all of them are able to perform buffer allocations.

3.2.2.2 Operations on buffers

From a functional point of view, VUMEM permits to clients to perform the following operations on buffers:

- Allocate/deallocate a buffer
- Access to a buffer allocated in another session
- Map/unmap a buffer in the current process address space
- Perform cache operations on buffers

3.2.3 Example of usage

3.2.3.1 Configuration tuning

It is possible to define the quantity of pdev allocated for VUMEM, thanks to the property `vumem-pdev`.

For instance, in vomx:

```
    vomx_be: vomx@be {                    // vOpenMX back-end for VM3

        compatible = "vomx";              //

      info        = "vumem-pdev=128K,",// vUMEM PDEV region size

                  "vrpq-reqs=256,",       // vRPQ requests number

                  "vrpq-pmem=256K";       // vRPQ PMEM region size

      #clone      = "auto";         // number of cloned vLINKs

      server;                       // server end point

   };
```

Like for VRPQ, this property has to be provided in the device tree property “info”.

This PDEV memory is used for internal driver routines. It will contain scattered lists of memory chunks, needing to be translated in case of sharing with other VMs. If this memory is not big enough, error messages will be raised. 128k is a good starting point.

Remark:

If for your own need a specific driver (like vomx) has to be created, and it must use VUMEM, you will have to get this property from the device tree. These modules performing inter VM communication are based on vlinks, providing APIs in order to get properties in the info field (like `vlink_param_size_get`).

3.2.3.2 Code example

Let’s see a skeleton of a typical code using VUMEM.

Let’s start by the server side!

Server side

First of all, initialization:

A VUMEM server has to be created, then the server waits for requests from clients:

```
    diag = vumemServerCreate(VUMEM_TEST_SRV_DEVICE, &server);

    while(1) {

           diag = vumemReceive(server, &req); //listen to requests
```

Where `VUMEM_TEST_SRV_DEVICE` is a string defining the VUMEM node to be used, for instance `vumem-srv-camera`. These node names are defined by vdrivers using the VUMEM back end (for server)/front end (for client) driver library.

The `req` variable will contain information about the user request.

Different types of client requests are identified thanks to a command id.

```
        //act depending on request type

     switch (req.header.cmd) {
```

the different possible command ids are defined in the header file server-lib.h. For instance, you can find:

- VUMEM_CMD_BUFFER_ALLOC: identify memory allocation requests, made for example by the client API `vumemBufferAlloc`
- VUMEM_CMD_BUFFER_FREE: for `vumemBufferFree`.

For instance, in order to process a memory allocation request:

```
case VUMEM_CMD_BUFFER_ALLOC:
      {
         //allocate memory
         …
}
```

The server will populate a variable of type `VumemResp`, containing information about the status of the request, and data about it (like the virtual address in the server space address of the allocated buffer, in case of buffer allocation request).

Once the server finishes its job, it will send its response to the client thanks to the routine `vumemRespond`:

```
diag = vumemRespond(server, &resp);
```

where `resp` is a variable of type `VumemResp`.

When the communication is finished, the server has to be destroyed with the following routine:

```
diag = vumemServerDestroy(server);
```

Remark: One important point to understand is that the VUMEM library has been designed to be generic. Basically, on the server side, memory can be allocated in many ways! Server does basically what it wants. It can call a memory allocation API provided by a proprietary library, a malloc, or even just provide a global buffer. The job of the VUMEM library will be to ensure that data are properly accessible from client sides.

Client side

Let's have a look to the client side:

First of all, it is required to create a session. This session can be a master one or not, implying different credentials about buffer allocation for other potential clients on the same session (please have a look to the API documentation for further details)

```
  diag = vumemSessionMasterCreate(VUMEM_TEST_CLT_DEVICE, &session);
```

Then, operations on buffer are possible. Let’s take the example of a memory allocation.

First of all, the routine `vumemBufferAlloc` has to be used:

```
    diag = vumemBufferAlloc(session,

                                 &alloc,

                                 sizeof(VumemAllocTest),

                                 &bufferHd);
```

The second parameter `alloc` is opaque for the routine. It has to contain all the required information by the server in order to perform the memory allocation, like the size of the buffer, or the kind of memory allocation. Of course, this type has to be known by both the client and the server.

Please notice that the server is able to modify `alloc`, permitting to have an additional communication channel between the client and the server.

The server can do it thanks to the following field of the `VumemResp` type (assuming type of `resp` is VumemResp):

```
resp.bufferAlloc.alloc
```

The third argument is the size of the variable `alloc`, and `bufferHd` a buffer handle that will permit to identify/access the allocated buffer of the current request.

Once the Buffer is allocated, it is necessary to map it in the address space of the client. It can be done with the following routine:

```
    diag = vumemBufferMap(bufferHd, (void**)&vaddr);
```

Then, vaddr points to the allocated buffer, it is now possible to access to it.

When work is finished, the following routines can be used in order to close connection:

```
    diag = vumemBufferUnmap(bufferHd); //unmap the buffer
    diag = vumemBufferFree(bufferHd); // free buffer allocated memory
    diag = vumemSessionDestroy(session); // destroy session
```

3.3 VBPIPE

3.3.1 Brief architecture considerations

Now let’s briefly describe virtual bi-directional pipe (vbpipe) user-level PV driver usage. The virtual bi-directional pipe driver offers byte-oriented, bi-directional communications between VMs. Given as an example, this driver runs on top of Linux, and can be adapted to run on top of real-time OS's. The virtual pipe mimics the behavior of Unix bi-directional pipes. It contains two unidirectional communication lines in opposite directions. Because of that, the vbpipe driver acts as a frontend as well as a backend driver.

The following figure describes the usual usage of the vbpipe user-level PV driver: initializing the communications, exchanging data through a dedicated vbpipe channel (between two VMs), and closing the communications.

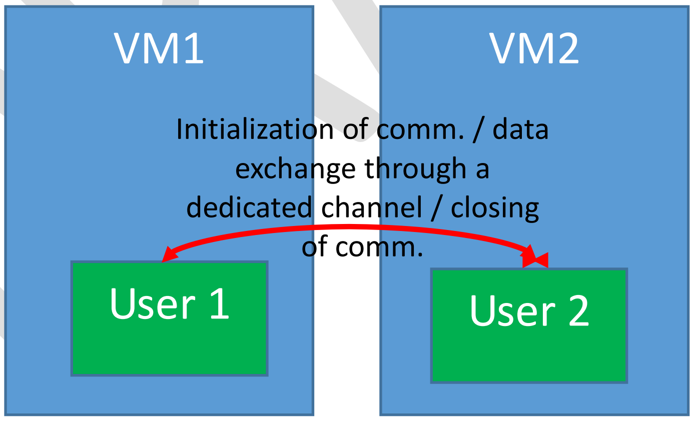
VM1
VM2
Initialization of comm. / data exchange through a dedicated channel / closing of comm.
User 1
User 2

Figure 5: vbpipe data transfer

3.3.2 How to use vbpipe

3.3.2.1 Configuration tuning

First of all, it is necessary to tune the VM configuration. To do this, we need to enable the vbpipe kernel driver in a VM by including the `CONFIG_VBPIPE` option into Linux kernel and add vlink nodes to the hypervisor device tree, one per endpoint. Below are sample vlink nodes for hypervisor device tree.

```
&vm2_vdevs {
    vbpipe24: vbpipe@be {                   // vbpipe back-end for VM4

        compatible = "vbpipe";              // uses generic vbpipe protocol

        info = ";;;wakeup=250;name=vbpipe0";

   };

};

&vm4_vdevs {

    vbpipe@fe {                             // vbpipe front-end

        peer-phandle = <&vbpipe24>;         // peer vLINK

        info = ";;;wakeup=250;name=vbpipe0";

    };

};
```

In the code listing above, a node compatible with `vbpipe` has to be added to the VM node having access to the vbpipe resources. The `vbpipe24` label is used to identify the same device (e.g., the vbpipe device here) and to reference two ends of the device found in two VMs. This has the effect of creating two vlink descriptors, allowing to have separate states in both directions.

Supposing the board named is `<board>`, the aforementioned code about vlink nodes can be added to the `<source_root>/nkernel-<vendor>/bsp/board/<SoC_family>/<board>/vdt-vcar.dts` device tree source file, which defines the hypervisor configuration for that board.

Let's have a look at a described code example. For a better readability, only little pieces of code are described, with partial checking status return. This is for brevity, but obviously something to avoid in a real-life code.

3.3.2.2 Sequence for vbpipe usage

The sequence listed below is followed in order to use vbpipe as shown in Figure 6:

1. Initialization of communications
   - "vbpipe user 1" opens a vbpipe device
   - "vbpipe user 2" opens a vbpipe device
2. Data exchanges
   - Read/write operations from/to the vbpipe
3. Closing of communications
   - Close vbpipes on both sides

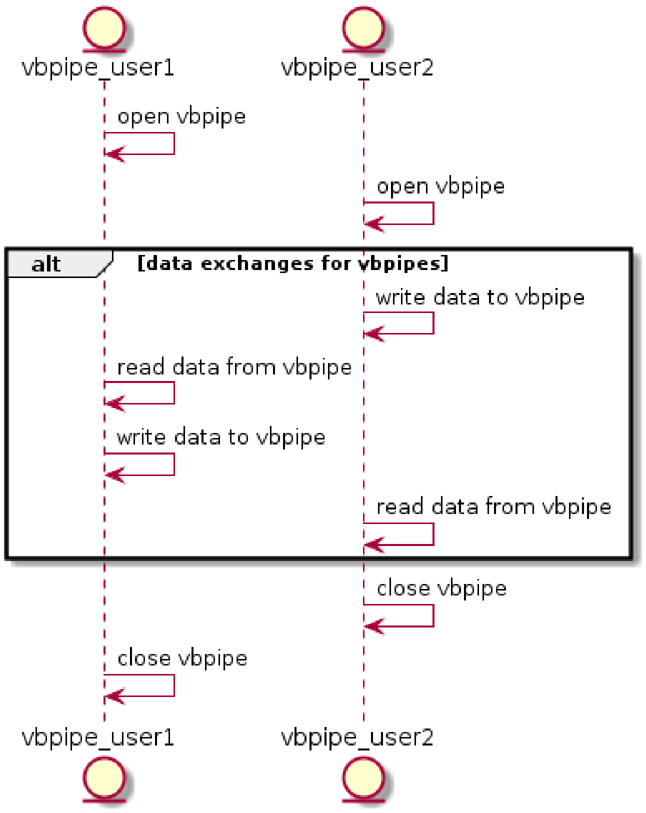

Figure 6: Sequence diagram of vbpipe user-level PV driver

3.3.2.3 Example of implementation

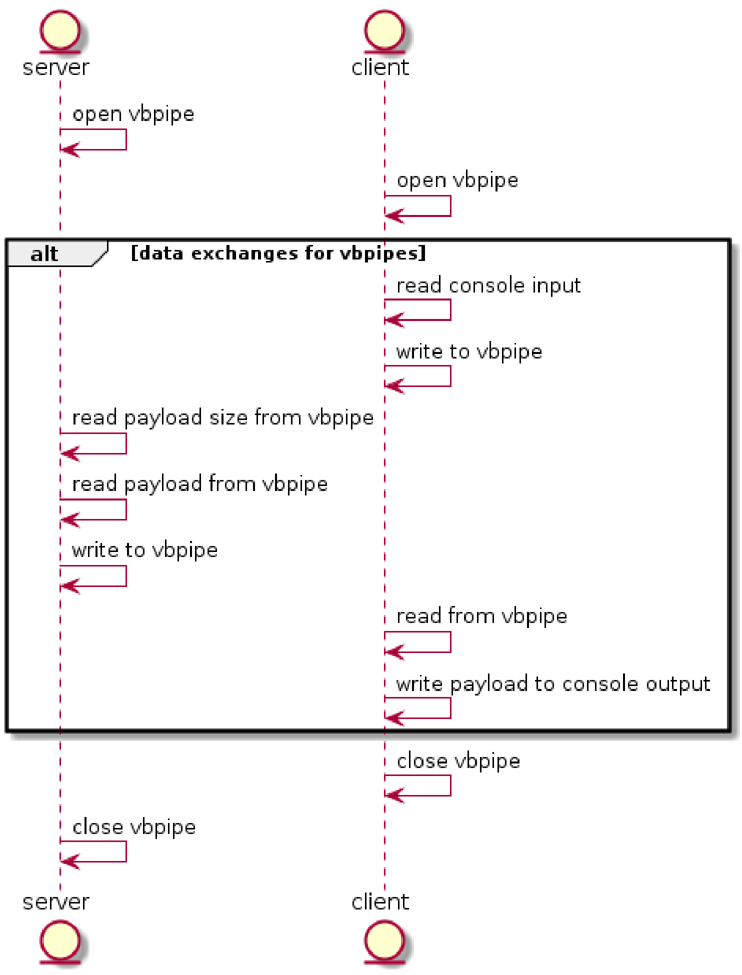

Figure 7: Sequence diagram of vbpipe user-level PV driver in hands-on

The example of implementation shown in Figure 7 is actually using standard POSIX API (please refer to “Linux Programmer’s Manual” for details).

- `int open(const char *pathname, int flags, mode_t mode)`

  The `open()` routine performs a rendez-vous between the ”vbpipe user 1” and the ”vbpipe user 2”. So opening a vbpipe device is blocking until the other hand opens the device.

- `ssize_t write(int fd, const void *buf, size_t count)`

The `write()` routine will block if the internal buffer of the vbpipe device is full or not big enough to store the data to be written. It will proceed in the other case. Once the reader has read the “vbpipe” the writer is unblocked.

- `ssize_t read(int fd, void *buf, size_t count)`

  The `read()` routine will block if the internal buffer of the vbpipe device has not enough data to fulfill the `read()` request. Then it will proceed if the other hand writes data in the internal buffer of the device.

- `int close(int fd)`

  The `close()` routine closes the file descriptor. It terminates the communications between two vbpipe users.

It should be noted that the `select()` routine is supported, though it is not present here. Feel free to read the `ex_poll()` API of the vbpipe kernel driver documentation for more information.

Now we will have a look at the following code, showing a simple usage of vbpipe, i.e.:

- Initialization of communications
- Exchanging data
- Closing of communications

Initialization of the communications

The first step for “vbpipe user 1” and “vbpipe user 2” is to initialize the communications with each other by opening a vbpipe device for accessing. The behavior of vbpipe mimics as close as possible the behavior of named pipes. Thus, the `open()` routine is blocking until the other hand has opened the communication. Below you can find example code, where the `mode` (with the value `0`) parameter is not used here.

```
  *fd = open(“/dev/vbpipe0”, O_RDWR, 0);
```

Listing 1. Server / client opens the vbpipe device

Exchanging data

At this moment, “vbpipe user 1” as the server tries to read from the vbpipe and to save the content to a char buffer, say `vbuf`. As the vbpipe internal buffer is empty now, so the read blocks.

```
  # vbuf: a pointer to a char buffer

  res = read(fd, vbuf, 1);

  if (res == 1) {

    res = vbuf[0];

    res = read(fd, vbuf + 1, res);

  }
```

Listing 2. Server reads the vbpipe device

Now “vbpipe user 2” as the client reads console input and stores them in a char buffer (say, `cbuf`), then saves the length of the payload at the beginning of the buffer.

```
sz = 1;

while ((ch = getchar()) != '\n' && ch != EOF) {

  cbuf[sz++] = ch;

}

cbuf[0] = sz - 1;
```

Listing 3. Client reads console input

Afterwards, the client writes the `cbuf` buffer address with the `sz` buffer size to the `fd` vbpipe file descriptor (see Listing 4 below). When the `write` returns -1, if the error number equals to `EPIPE`, it indicates that the server closes the connection; otherwise, it is a client error. If the vbpipe buffer becomes full, the write will block till the vbpipe buffer is read by the server.

```
res = write(fd, cbuf, sz);

if (res == -1) {

  if (errno == EPIPE) {

    error("server closed connection\n");

    return -1;

  } else {

    error("client error\n");

    return -1;

  }

} else {

  /* See Listing 6 */

}
```

Listing 4. Client writes to the vbpipe device

When the write succeeds, the read by the server (see Listing 2) will no longer block. If the read returns a negative value, it indicates an unexpected error; otherwise, the read succeeds and it will write what it reads to the `fd` file descriptor. If the write returns a negative value and the error number is `EPIPE,` it means an unexpected write error.

```
if (res < 0) {
    error("unexpected read error\n");

    return -1;

  } else if (res > 0) {

    res = write(fd, vbuf, res + 1);

    if (res < 0 && errno != EPIPE) {

      error("unexpected write error\n");

      return -1;

    }

  }
```

Listing 5. Server writes to the vbpipe device

When the write succeeds in Listing 4, the client will read the `fd` file descriptor. If the read returns a non-positive value, it means the server closes the connection; otherwise, it will write the payload of the read to console output.

```
  if (res == -1) {

    …

  } else {

    res = read(fd, vbuf, sz);

    if (res < 0 || res == 0) {

      error("server closed connection\n");

      return -1;

    } else {

      (void) write(1, vbuf + 1, sz);

    }

  }
```

Listing 6. Client reads the vbpipe device and writes to console output

Closing of communications

When the client completes reading vbpipe, it closes the used vbpipe file descriptor by calling the `close()` routine. The same does the server.

```
  close(fd);
```