HARMAN

Virtual Device Driver Reference Manual - v12.1
Public Edition

Build 0041

Generated by Doxygen 1.8.14

Contents

Chapter 1

Manual Pages

The collection of virtual device driver manual pages is divided into several sections:

Section name	Summary
(3D) NKDDI	The Hypervisor Device Driver Interface is used by front-end and back-end drivers, and guest operating system adaptations
(4D) vdrivers	Existing virtual drivers running on top of the Hypervisor inside guest operating systems
(5) Files and directories	Files and directories managed by existing virtual drivers
(8) Administration and privileged commands	Special programs, e.g. daemons, used with virtual drivers

1.1 (3D) NKDDI

NK_ATOMIC_ADD(3D)

NK_ATOMIC_CLEAR(3D)

NK_ATOMIC_SET(3D)

NK_ATOMIC_SUB(3D)

NK_BALLOON_CTRL(3D)

NK_BIT2MASK(3D)

NK_CLEAR_AND_TEST(3D)

NK_CONS_HIST_BOOT_ORDINAL(3D)

NK_CONS_HIST_GETCHAR(3D)

NK_CONS_POLL(3D)

NK_CONS_WRITE(3D)

NK_HV_VERSION_GET(3D)

NK_ID_GET(3D)

1.1.1 NK_ATOMIC_ADD(3D)

1.1.1.1 NAME

nk_atomic_add — Performs atomic addition

1.1.1.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_atomic_add**(nku32_f∗ *ptr*, nku32_f *val*);

1.1.1.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The arithmetical operation ∗**ptr += val** is performed atomically.

1.1.1.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.1.4 SEE ALSO

nk_mask2bit

nk_bit2mask

nk_atomic_set

nk_atomic_clear

nk_clear_and_test

nk_atomic_sub

nk_sub_and_test

1.1.2 NK_ATOMIC_CLEAR(3D)

1.1.2.1 NAME

nk_atomic_clear — Clears bit mask atomically

1.1.2.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_atomic_clear**(nku32_f∗ *mask*, nku32_f *clear*);

1.1.2.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The logical operation ∗**mask &=** ∼**clear** is performed atomically.

1.1.2.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.2.4 SEE ALSO

nk_mask2bit

nk_bit2mask

nk_atomic_set

nk_clear_and_test

nk_atomic_add

nk_atomic_sub

nk_sub_and_test

1.1.3 NK_ATOMIC_SET(3D)

1.1.3.1 nk_atomic_set_NAME

nk_atomic_set — Sets bit mask atomically

1.1.3.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_atomic_set**(nku32_f∗ *mask*, nku32_f *set*);

1.1.3.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The logical operation ∗**mask** |**= set** is performed atomically.

1.1.3.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.3.4 SEE ALSO

nk_mask2bit

nk_bit2mask

nk_atomic_clear

nk_clear_and_test

nk_atomic_add

nk_atomic_sub

nk_sub_and_test

1.1.4 NK_ATOMIC_SUB(3D)

1.1.4.1 NAME

nk_atomic_sub — Performs atomic subtraction

1.1.4.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_atomic_sub**(nku32_f∗ *ptr*, nku32_f *val*);

1.1.4.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The arithmetical operation ∗**ptr -= val** is performed atomically.

1.1.4.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.4.4 SEE ALSO

nk_mask2bit

nk_bit2mask

nk_atomic_set

nk_atomic_clear

nk_clear_and_test

nk_atomic_add

nk_sub_and_test

1.1.5 NK_BALLOON_CTRL(3D)

1.1.5.1 NAME

nk_balloon_ctrl — Perform a memory ballooning control operation

1.1.5.2 SYNOPSIS

#include <nk/nkern.h>

nku32_f **nkops.nk_balloon_ctrl**(int *op*, nku32_f∗ *pfns*, nku32_f *count*);

1.1.5.3 DESCRIPTION

This primitive is invoked to perform a memory ballooning control operation. Memory ballooning must be enabled and configured in the Hypervisor.

The first *op* parameter contains the requested operation. Allowed operations are listed below:

NK_BALLOON_STATUS - This operation returns the current status of the memory ballooning for the calling VM.

NK_BALLOON_HOST - This operation gets the balloon host VM ID, the VM that initially hosts all physical memory available for ballooning.

NK_BALLOON_RGN_BASE - This operation gets the balloon region base address in PFNs.

There are two kind of VMs that participate in memory ballooning:

- one host VM which initially hosts and potentially uses all physical memory available for ballooning.
- a set of non-host VMs that may receive physical memory from memory ballooning

For each non-host VM, the Hypervisor provisions a region capable to receive any subset of physical pages available for ballooning in the VM's physical address (IPA) space.

NK_BALLOON_RGN_SIZE - This operation gets the balloon region size in PFNs.

NK_BALLOON_ALLOC - This operation allows to allocate a list of pages from the Hypervisor memory region reserved for memory ballooning for the current VM memory space. This newly allocated memory may have been previously released to the Hypervisor by another VM.

NK_BALLOON_FREE - This operation allows to free a list of pages previously allocated for ballooning to the Hypervisor, so it can be reallocated by other VMs.

The second *frames* parameter is a pointer to a table of page frame numbers.

The third *count* parameter is the count of requested page frames in frames array.

These two last parameters are only valid for *NK_BALLOON_ALLOC* and *NK_BALLOON_FREE* operations, and set to 0 for other operations.

1.1.5.3.1 RETURN VALUES

This primitive returns the following values according to the requested command:

NK_BALLOON_ENABLED

if the requested operation is *NK_BALLOON_STATUS* and if the memory ballooning is enabled for the calling VM;

NK_BALLOON_DISABLED

if the requested operation is *NK_BALLOON_STATUS* and if the memory ballooning is disabled for the calling VM. This error is returned if the memory ballooning is really disabled for this VM, or if the memory ballooning is not enabled in the Hypervisor. In that latter case, this result is returned for all VMs.

This primitive returns the count of page frames allocated for *NK_BALLOON_ALLOC* operations, and the page frames freed for *NK_BALLOON_FREE* operations. In case of memory ballooning exhaustion for the *NK_BALLOON_ALLOC* operation, this primitive returns 0.

1.1.6 NK_BIT2MASK(3D)

1.1.6.1 NAME

nk_bit2mask — Returns a bit mask with only one bit set, others bits are 0

1.1.6.2 SYNOPSIS

#include <nk/nkern.h>

nku32_f **nkops.nk_bit2mask**(nku32_f *bit*);

1.1.6.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The bits in the bit mask (bit set) are indexed from 0 to 31. The bit 0 is considered as the highest priority bit, the bit 31 is the lowest priority bit. Note that the actual in-memory bit mask representation is processor dependent and may use a different bit indexing schema.

1.1.6.3.1 RETURN VALUES

This primitive returns the bit mask with only bit number bit set, others bits are 0. The caller must ensure bit value is in range 0..31.

1.1.6.4 SEE ALSO

nk_mask2bit

nk_atomic_set

nk_atomic_clear

nk_clear_and_test

nk_atomic_add

nk_atomic_sub

nk_sub_and_test

1.1.7 NK_CLEAR_AND_TEST(3D)

1.1.7.1 NAME

nk_clear_and_test — Clears bit mask atomically and tests

1.1.7.2 SYNOPSIS

#include <nk/nkern.h>

nku32_f **nkops.nk_clear_and_test**(nku32_f∗ *mask*, nku32_f *clear*);

1.1.7.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The logical operation ∗**mask &=** ∼**clear** is atomically performed.

1.1.7.3.1 RETURN VALUES

The function returns zero if and only if the result of the operation is zero.

1.1.7.4 SEE ALSO

nk_mask2bit

nk_bit2mask

nk_atomic_set

nk_atomic_clear

nk_atomic_add

nk_atomic_sub

nk_sub_and_test

1.1.8 NK_CONS_HIST_BOOT_ORDINAL(3D)

1.1.8.1 NAME

nk_cons_hist_boot_ordinal — Get the ordinal number of the first character written in the history buffer after reboot

1.1.8.2 SYNOPSIS

#include <nk/nkern.h>

nku64_f **nk_con_hist_boot_ordinal**(void);

1.1.8.3 DESCRIPTION

This function allows to distinguish in the persistent history buffer the content written before reboot from the content written after reboot: all the characters written in the buffer before reboot get ordinals lower than the ordinal returned by this function, whereas all the characters written after reboot get ordinals higher or equal.

1.1.8.3.1 RETURN VALUES

This primitive returns the ordinal number of the first character that has been written (or will be written) in the history buffer after reboot.

1.1.8.4 SEE ALSO

nk_cons_hist_getchar

1.1.9 NK_CONS_HIST_GETCHAR(3D)

1.1.9.1 NAME

nk_cons_hist_getchar — Get one character from the history buffer

1.1.9.2 SYNOPSIS

#include <nk/nkern.h>

int **nk_con_hist_getchar**(nku64_f∗ ordnp);

1.1.9.3 DESCRIPTION

The first available character stored in the history buffer with the ordinal number equal or higher than the argument ordinal number is returned. In addition, the ordinal number of the next character is also returned. This number corresponds to a character which may be already written in the buffer or not.

The *ordnp* parameter is a pointer to the ordinal number of the requested character.

1.1.9.3.1 RETURN VALUES

This primitive returns the ASCII code of the requested character or -1 if there is no available character. The ordinal number of the next character is also returned in *ordnp*.

1.1.9.4 SEE ALSO

nk_cons_hist_boot_ordinal

1.1.10 NK_CONS_POLL(3D)

1.1.10.1 NAME

nk_cons_poll — Poll the serial console input

1.1.10.2 SYNOPSIS

#include <nk/nkern.h>

int **nk_con_poll**(void);

1.1.10.3 DESCRIPTION

This primitive polls the Hypervisor serial console input and returns either a received character or -1. It can be used together with nk_cons_write in order to implement a virtual console driver in a VM running on top of the Hypervisor.

Such a virtual console driver might be an attractive alternative to an emulated UART device connected to the Hypervisor console.

1.1.10.3.1 RETURN VALUES

This primitive returns the ASCII code of a pending character or -1 if there is no pending character.

1.1.10.4 SEE ALSO

nk_cons_write

1.1.11 NK_CONS_WRITE(3D)

1.1.11.1 NAME

nk_cons_write — Put characters to the console

1.1.11.2 SYNOPSIS

#include <nk/nkern.h>

void **nk_cons_write**(const char∗ buf, int size);

1.1.11.3 DESCRIPTION

This primitive writes a buffer of characters to the Hypervisor serial console. It can be used together with nk_cons_poll in order to implement a virtual console driver in a VM running on top of the Hypervisor.

Such a virtual console driver might be an attractive alternative to an emulated UART device connected to the Hypervisor console.

The *buf* argument points to a buffer which contains characters to be written.

The *size* argument specifies the size of the buffer in bytes.

1.1.11.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.11.4 SEE ALSO

nk_cons_poll

1.1.12 NK_HV_VERSION_GET(3D)

1.1.12.1 NAME

nk_hv_version_get — Parse and return the components of the hypervisor interface version.

1.1.12.2 SYNOPSIS

#include <nk/nkern.h>

int **nkops.nk_hv_version_get**(unsigned int∗ *major*, unsigned int∗ *minor*);

1.1.12.3 DESCRIPTION

This primitive retrieves and parses the hypervisor API version, returning the *major* and *minor* component of the API version through the respective arguments.

The hypervisor API version is retrieved through the *nk.version.api* property with the `nk_prop_get`**(3D)** primitive. The version itself is in the *"MAJOR.MINOR.PATCH"* format.

1.1.12.3.1 VALUES

Zero is returned if the version has been parsed correctly, otherwise -1 is returned.

1.1.12.4 SEE ALSO

`nk_prop_get`**(3D)**

1.1.13 NK_ID_GET(3D)

1.1.13.1 NAME

nk_id_get — Reports current VM identifier

1.1.13.2 SYNOPSIS

#include <nk/nkern.h>

NkOsId **nkops.nk_id_get**();

1.1.13.3 DESCRIPTION

The function reports the current VM identifier in a *NkOsId* object. No parameter is provided by the caller. Identifiers 0 and 1 are reserved by the Hypervisor, so VM identifiers are 2, 3, and so forth.

1.1.13.3.1 RETURN VALUES

The current VM identifier is returned by this primitive.

1.1.13.4 SEE ALSO

nk_last_id_get

nk_running_ids_get

1.1.14 NK_LAST_ID_GET(3D)

1.1.14.1 NAME

nk_last_id_get — Reports last running VM identifier

1.1.14.2 SYNOPSIS

#include <nk/nkern.h>

NkOsId **nkops.nk_last_id_get**();

1.1.14.3 DESCRIPTION

This function reports the last running VM identifier in a *NkOsId* object.

1.1.14.3.1 RETURN VALUES

The last running VM identifier is returned by this primitive.

1.1.14.4 SEE ALSO

nk_id_get

nk_running_ids_get

1.1.15 NK_MASK2BIT(3D)

1.1.15.1 NAME

nk_mask2bit — Returns the mask with only the highest priority bit set

1.1.15.2 SYNOPSIS

#include <nk/nkern.h>

nku32_f **nkops.nk_mask2bit**(nku32_f *mask*);

1.1.15.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The unique *mask* parameter holds a non-zero mask whose bit(s) set are indexed from 0 to 31. The bit 0 is considered as the highest priority bit, the bit 31 is the lowest priority bit. Note that the actual in memory bit mask representation is processor dependent (that is, endianess) and may use different bit indexing schema.

1.1.15.3.1 RETURN VALUES

This primitive returns the index of the highest priority bit set. The caller must ensure that the mask is not set to zero.

1.1.15.4 SEE ALSO

nk_bit2mask

nk_atomic_set

nk_atomic_clear

nk_clear_and_test

nk_atomic_add

nk_atomic_sub

nk_sub_and_test

1.1.16 NK_MEM_MAP(3D)

1.1.16.1 NAME

nk_mem_map — Map a chunk of persistent physical memory

1.1.16.2 SYNOPSIS

#include <nk/nkern.h>

void∗ **nkops.nk_mem_map**(NkPhAddr *paddr*, NkPhSize *size*);

1.1.16.3 DESCRIPTION

The nk_mem_map primitive invokes standard OS services to map a chunk of persistent physical memory obtained from nk_pmem_alloc.

The first *paddr* parameter is a valid physical address belonging to the current VM.

The second *size* parameter is the size to map expressed in bytes.

1.1.16.3.1 RETURN VALUES

This primitive returns the virtual address of the mapped area, otherwise zero is returned.

1.1.16.4 SEE ALSO

nk_mem_unmap

nk_pmem_alloc

1.1.17 NK_MEM_UNMAP(3D)

1.1.17.1 NAME

nk_mem_unmap — Unmap a chunk of persistent physical memory

1.1.17.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_mem_unmap**(void∗ *vaddr*, NkPhAddr *paddr*, NkPhSize *size*);

1.1.17.3 DESCRIPTION

The nk_mem_unmap primitive unmaps a chunk of persistent physical memory previously mapped by the nk_mem_map.

The first *vaddr* parameter is a valid virtual address belonging to the current VM and previously returned by **nk_mem_map(3D)**.

The second *paddr* is the physical address corresponding to *vaddr* and belonging to the current VM.

The last *size* parameter is the size to unmap expressed in bytes.

1.1.17.3.1 RETURN VALUES

No relevant value is returned by this primitive

1.1.17.4 SEE ALSO

nk_mem_map

1.1.18 NK_PDEV_ALLOC(3D)

1.1.18.1 NAME

nk_pdev_alloc — Allocates memory from the Hypervisor persistent device repository

1.1.18.2 SYNOPSIS

#include <nk/nkern.h>

NkPhAddr **nkops.nk_pdev_alloc**(NkPhAddr *vlink*, NkResourceId *id*, NkPhSize *size*);

1.1.18.3 DESCRIPTION

This primitive allocates a chunk of memory from the Hypervisor persistent device repository.

The first *vlink* parameter is a physical address of the virtual link associated with the virtual device descriptor in the Hypervisor persistent device repository (see the nk_vlink_lookup manual page for further details).

The second *id* parameter is a memory chunk identifier (0, 1, 2, and so on).

The last *size* parameter is the size expressed in bytes to be allocated in the Hypervisor persistent device repository.

The allocated memory block is labeled using the couple *vlink*, *id*. It is guaranteed to get a unique memory block for a unique label. Consequently, multiple invocations with the same label yield the same result. This allows to support VM reboots.

1.1.18.3.1 RETURN VALUES

This primitive returns the physical address of allocated chunk belonging to the Hypervisor persistent device repository in case of success, otherwise zero is returned.

The return value is a physical address and must be converted to a virtual one before usage by invoking the nk_ptov primitive.

1.1.18.4 SEE ALSO

nk_vlink_lookup

nk_ptov

nk_vtop

1.1.19 NK_PMEM_ALLOC(3D)

1.1.19.1 NAME

nk_pmem_alloc — Allocate contiguous memory from the global persistent memory pool

1.1.19.2 SYNOPSIS

#include <nk/nkern.h>

NkPhAddr **nkops.nk_pmem_alloc**(NkPhAddr *vlink*, NkResourceId *id*, NkPhSize *size*);

1.1.19.3 DESCRIPTION

This primitive allocates a chunk of contiguous memory from the global persistent memory. This memory is used as a shared area between frontend and backend drivers running on different VMs.

The first *vlink* parameter is the physical address of the virtual link device the allocated memory chunk will be associated with (see the nk_vlink_lookup manual page for further details).

The second *id* parameter is a memory chunk identifier (0, 1, 2, and so on).

The last *size* parameter is the size expressed in bytes to be allocated in the Hypervisor persistent repository.

The allocated memory is rounded to the nearest multiple of page size. The allocated memory block is labeled using the couple of arguments: *vlink* and *id*. A unique couple of *vlink* and *id* yields a unique allocated memory block. Consequently, multiple invocations with the same values for *vlink* and *id* always return the same result. This allows to support VM reboots.

1.1.19.3.1 RETURN VALUES

In case of success, this primitive returns the physical base address of the allocated memory aligned on page boundary. In case of error, 0 is returned.

A physical address is returned by this primitive and must be mapped into the current virtual space through the nk_mem_map primitive.

1.1.19.4 SEE ALSO

nk_mem_map

nk_mem_unmap

1.1.20 NK_PROP_ENUM(3D)

1.1.20.1 NAME

nk_prop_enum — Get (name,value) of a property

1.1.20.2 SYNOPSIS

#include <nk/nkern.h>

int **nkops.nk_prop_enum** (unsigned int *pid*, char∗ *name*, unsigned int *nlen_max*, NkPropAttr∗ *attr*);

1.1.20.3 DESCRIPTION

This primitive returns the name and attributes of the property specified by the *pid* argument.

The *name* argument points to a buffer to which the property name is copied.

The *nlen_max* argument specifies the size of the name buffer in bytes.

The *attr* argument specifies a 32 bit word address where the property attributes are returned. The property attributes are combined from the permissions and the real name length (including the terminating zero character).

1.1.20.3.1 RETURN VALUES

On error, a negative error code is returned as described below:

NK_PROP_UNKNOWN

This error is returned when the property name is not found in the Hypervisor repository of properties.

NK_PROP_ERROR

This error is returned on any other error such as invalid memory addresses.

On success, the maximum property value size is returned.

1.1.20.4 SEE ALSO

nk_prop_set

nk_prop_get

1.1.21 NK_PROP_GET(3D)

1.1.21.1 NAME

nk_prop_get — Read the value of a given Hypervisor property

1.1.21.2 SYNOPSIS

#include <nk/nkern.h>

int **nkops.nk_prop_get**(char∗ *name*, void∗ *value*, unsigned int *maxsize*);

1.1.21.3 DESCRIPTION

This primitive reads the value of a given property.

The *name* argument points to a null terminated ASCII string specifying the property name.

The *value* argument points to a buffer to which the property value is copied.

The *maxsize* argument specifies the buffer size.

1.1.21.3.1 RETURN VALUES

On error, a negative error code is returned as described below:

NK_PROP_BUSY

This error is returned when a property update is in progress. In other words, when a contention with a nk_prop_set call is detected. As a common rule, receiving such an error, the VM has to try again after some reasonable delay. An important exception from the above rule is a nk_prop_get primitive issued in the *NK_↩ XIRQ_SYSCONF* interrupt handler as a reaction on a property update event. Upon receiving such an error in the **NK_XIRQ_SYSCONF** handler, the software should simply return from the interrupt handler. The Hypervisor guarantees that a **NK_XIRQ_SYSCONF** cross interrupt will be sent again once the update in progress is finished.

NK_PROP_PERMISSION

This error is returned when the operation is disabled for this VM by the property permissions.

NK_PROP_UNKNOWN

This error is returned when the property name is not found in the Hypervisor repository of properties.

NK_PROP_ERROR

This error is returned on any other error such as invalid memory addresses.

On success, the number of bytes which have been effectively copied to the buffer is returned. When the current size of the property value exceeds the buffer size, only first *max_size* bytes are copied.

1.1.21.4 SEE ALSO

nk_prop_set

nk_prop_enum

1.1.22 NK_PROP_SET(3D)

1.1.22.1 NAME

nk_prop_set — Read the value of a given Hypervisor property

1.1.22.2 SYNOPSIS

#include <nk/nkern.h>

int **nkops.nk_prop_set**(char∗ *name*, void∗ *value*, unsigned int *size*);

1.1.22.3 DESCRIPTION

This primitive updates the value of a given property and sends a notification event.

The *name* argument points to a null terminated ASCII string specifying the property name.

The *value* argument points to the new property value.

The *size* argument specifies the new property value size.

In addition, this primitive sends a notification event through the *NK_XIRQ_SYSCONF* cross interrupt to all running VMs which are allowed to receive this event.

1.1.22.3.1 RETURN VALUES

On error, a negative error code is returned as described below:

NK_PROP_BUSY

This error is returned when a property update is in progress. In other words, when a contention with another nk_prop_set call is detected. As a common rule, receiving such an error, the VM has to try again after some reasonable delay. An important exception from the above rule is a nk_prop_set primitive issued in the *NK_XIRQ_SYSCONF* interrupt handler as a reaction on a property update event. Upon receiving such an error in the **NK_XIRQ_SYSCONF** handler, the software should simply return from the interrupt handler. The Hypervisor guarantees that a **NK_XIRQ_SYSCONF** cross interrupt will be sent again once the update in progress is finished.

NK_PROP_PERMISSION

This error is returned when the operation is disabled for this VM by the property permissions.

NK_PROP_UNKNOWN

This error is returned when the property name is not found in the Hypervisor repository of properties.

NK_PROP_ERROR

This error is returned on any other error such as invalid memory addresses.

On success, the number of bytes which have been effectively copied to the property value is returned. The returned value size can be less than the requested size when the last one exceeds the maximum value size.

1.1.22.4 SEE ALSO

nk_prop_get

nk_prop_enum

1.1.23 NK_PTOV(3D)

1.1.23.1 NAME

nk_ptov — Return the virtual address corresponding to a physical address

1.1.23.2 SYNOPSIS

#include <nk/nkern.h>

void∗ **nkops.nk_ptov**(NkPhAddr *paddr*);

1.1.23.3 DESCRIPTION

The Hypervisor supports processors with MMU. On such platforms operating systems may use different mappings for the physical memory. Because of that virtual addresses cannot be used as a pointers in the data structures accessed from different operating systems. The physical addresses are used in this case. The NKDDI provides address translation functions nk_vtop, nk_ptov used for physical addresses belonging to the global persistent repository where virtual device descriptors are stored.

The nk_ptov primitive translates a physical memory address held in the *paddr* parameter and belonging to the global persistent device repository into a virtual address. Typically, physical address returned by nk_pdev_alloc primitive is translated into virtual address through nk_ptov.

1.1.23.3.1 RETURN VALUES

This primitive returns the virtual address corresponding to the physical address belonging to the global persistent device repository. In case of error, the operating system kernel will be panicked; BUG() is issued on Linux.

1.1.23.4 SEE ALSO

nk_vtop

nk_pdev_alloc

1.1.24 NK_PXIRQ_ALLOC(3D)

1.1.24.1 NAME

nk_pxirq_alloc — Allocate a contiguous range of free persistent cross interrupts

1.1.24.2 SYNOPSIS

#include <nk/nkern.h>

NkXIrq **nkops.nk_pxirq_alloc**(NkPhAddr *vlink*, NkResourceId *id*, NkOsId *osid*, int *nb*);

1.1.24.3 DESCRIPTION

The first *vlink* parameter is a valid physical address with the virtual link device associated with a virtual device descriptor in the repository (see the nk_vlink_lookup manual page for further details).

The second *id* parameter is a valid virtual link identifier (0, 1, 2, and so on).

The third *osid* parameter is a valid VM identifier.

The last *nb* parameter is the number of persistent cross interrupts to allocate.

This primitive allocates a contiguous range of *nb* persistent free cross interrupts for a given VM identifier *osid*. The allocated persistent free range of cross interrupts is labeled using the *vlink* and *id* couple of parameters. It is guaranteed that a unique label (for example, couple of *vlink*, *id*) yields a unique persistent cross interrupts range. Consequently, multiple invocations with the same label always return the same result. This allows to support reboot of a VM running for example a frontend driver. This is the main reason why this primitive must be used instead of the deprecated nk_xirq_alloc primitive.

1.1.24.3.1 RETURN VALUES

This primitive returns the first persistent cross interrupt number if allocation is successful. In case of error, a null value is returned.

1.1.24.4 SEE ALSO

nk_xirq_attach

nk_xirq_detach

nk_xirq_mask

nk_xirq_ummask

nk_xirq_affinity

nk_xirq_trigger

1.1.25 NK_RESUME(3D)

1.1.25.1 NAME

nk_resume — Resume a VM

1.1.25.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_resume**(NkOsId *osid*);

1.1.25.3 DESCRIPTION

This primitive invokes the nk_prop_set to resume a VM: the nk.vm.*osid*.state.paused property is set to 0.

The parameter *osid* gives the identifier of the resumed VM.

1.1.25.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.25.4 SEE ALSO

nk_id_get

nk_last_id_get

nk_running_ids_get

nk_stop

nk_suspend

nk_start

nk_prop_set

1.1.26 NK_RUNNING_IDS_GET(3D)

1.1.26.1 NAME

nk_running_ids_get — Reports the bit mask of currently running VMs

1.1.26.2 SYNOPSIS

#include <nk/nkern.h>

NkOsMask **nkops.nk_running_ids_get**(void);

1.1.26.3 DESCRIPTION

This primitive reports the bit mask of currently running VMs, all others are either stopped or non present.

1.1.26.3.1 RETURN VALUES

A bitmask is returned by this routine containing all running VM identifiers. The index of each set bit is the running VM identifier.

1.1.26.4 SEE ALSO

nk_id_get

nk_last_id_get

1.1.27 NK_START(3D)

1.1.27.1 NAME

nk_start — Start another VM or restart current VM

1.1.27.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_start**(NkOsId *osid*);

1.1.27.3 DESCRIPTION

This primitive invokes the nk_prop_set to start a VM: the nk.vm.*osid*.state.running property is set to 1.

The parameter *osid* gives the identifier of the started VM.

When *osid* matches the current VM, the current VM is restarted. Indeed, it is not possible to chain nk_stop and nk_start on oneself, as the first call would never return.

1.1.27.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.27.4 SEE ALSO

nk_id_get

nk_last_id_get

nk_running_ids_get

nk_stop

nk_suspend

nk_resume

nk_prop_set

1.1.28 NK_STOP(3D)

1.1.28.1 NAME

nk_stop — Stop a VM

1.1.28.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_stop**(NkOsId *osid*);

1.1.28.3 DESCRIPTION

This primitive invokes the nk_prop_set to stop a VM: the nk.vm.*osid*.state.running property is set to 0.

The parameter *osid* gives the identifier of the stopped VM.

1.1.28.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.28.4 SEE ALSO

nk_id_get

nk_last_id_get

nk_running_ids_get

nk_start

nk_suspend

nk_resume

nk_prop_set

1.1.29 NK_SUB_AND_TEST(3D)

1.1.29.1 NAME

nk_sub_and_test — Performs atomic subtraction and test

1.1.29.2 SYNOPSIS

#include <nk/nkern.h>

nku32_f **nkops.nk_sub_and_test** (nku32_f∗ *ptr*, nku32_f *val*);

1.1.29.3 DESCRIPTION

The 32 bit unsigned integers and 32 bit masks (bit sets) are data types frequently used in the virtual device drivers. In most cases, an atomic operation on those data types can be used to avoid more complicated synchronization methods between frontend and backend drivers.

The arithmetical operation ∗**ptr -= val** is atomically performed.

1.1.29.3.1 RETURN VALUES

This primitive returns zero if and only if the result of the operation is zero.

1.1.29.4 SEE ALSO

nk_mask2bit

nk_bit2mask

nk_atomic_set

nk_atomic_clear

nk_clear_and_test

nk_atomic_add

nk_atomic_sub

1.1.30 NK_SUSPEND(3D)

1.1.30.1 NAME

nk_suspend — Suspend a VM

1.1.30.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_suspend**(NkOsId *osid*);

1.1.30.3 DESCRIPTION

This primitive invokes the nk_prop_set to suspend a VM: the nk.vm.*osid*.state.paused property is set to 1.

The parameter *osid* gives the identifier of the suspended VM.

1.1.30.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.30.4 SEE ALSO

nk_id_get

nk_last_id_get

nk_running_ids_get

nk_stop

nk_start

nk_resume

nk_prop_set

1.1.31 NK_VLINK_LOOKUP(3D)

1.1.31.1 NAME

nk_vlink_lookup — Search for a vlink device in the persistent device repository

1.1.31.2 SYNOPSIS

#include <nk/nkern.h>

NkPhAddr **nkops.nk_vlink_lookup**(const char∗ *name*, NkPhAddr *plink*);

1.1.31.3 DESCRIPTION

When there are multiple VMs running on the same board, communication must be achieved between all of them. Pairs of drivers (frontend and backend) located in each VM allow to exchange data between VMs. Such pairs of drivers use a global persistent shared memory for large chunks of memory, and a persistent device repository for devices descriptors which are accessible from both VMs. This persistent device repository is considered as a virtual communication link and described by a dedicated *NkDevVlink* object whose fields are shown below:

```
@#define NK_DEV_VLINK_OFF   0
@#define NK_DEV_VLINK_RESET 1
@#define NK_DEV_VLINK_ON    2

@#define NK_DEV_VLINK_NAME_LIMIT    16

typedef struct NkDevVlink {
    char        name[NK_DEV_VLINK_NAME_LIMIT];
                                /* name of communication link */
                                /* (actually virtual device class, */
                                /* like veth, vbd, etc.) */
                                /* max 15 characters + ending zero */
    int             link;       /* global/unique communication link number */
    NkOsId          s_id;       /* server VM id */
    volatile int    s_state;    /* server VM state: off, reset, on */
    nku32_f         s_info;     /* server specific info */
    NkOsId          c_id;       /* client VM id */
    volatile int    c_state;    /* client VM state: off, reset, on */
    nku32_f         c_info;     /* client specific info */
    nku32_f         pad0;
    nku64_f         pad1;
} NkDevVlink;
```

All virtual link (vlink) descriptors are created by the Hypervisor. The current implementation parses the VM device tree to find vlink descriptions and creates corresponding data structures.

A virtual communication link is an asymmetric peer to peer link. On one side, there is a server, on another one side a client.

name is a name of a virtual communication link (actually a virtual device class like veth, vaudio, and so on) with a maximum of 15 characters and ending with NUL.

link is a unique/global virtual link number for the vlinks with the same *name*. It is used to differentiate those vlinks.

s_id is a server VM identifier (VM where server/backend driver is located).

s_info is a server specific information — physical address of a character string with some information specific for each driver. This address is located in PDEV memory, so the nk_ptov call has to be used to access it in virtual address space.

c_id is a client VM identifier (VM where client/frontend driver is located).

c_info is a client specific information — physical address of a character string with some information specific for each driver. This address is located in PDEV memory, so the nk_ptov call has to be used to access it in virtual address space.

s_state is a state of server driver.

c_state is a state of client driver.

The state is used to implement a handshake during driver initialization. It reflects different stages of driver readiness:

NK_DEV_VLINK_OFF state means that the driver is not ready to communicate with its peer counterpart.

NK_DEV_VLINK_RESET state means that the driver has finished its internal/local initialization. All variables owned by this driver and visible on both sides are initialized. To complete initialization, a driver might need to read variables owned by its counterpart. It can do only so when its peer driver is in *NK_DEV_VLINK_RESET* or *NK_DEV_VLIN←K_ON* state.

NK_DEV_VLINK_ON state means that the driver is ready to communicate with its peer counterpart.

The driver is allowed to change its state to *NK_DEV_VLINK_ON* only from *NK_DEV_VLINK_RESET* state and only when its counterpart is in *NK_DEV_VLINK_RESET* or *NK_DEV_VLINK_ON* state.

If a driver detects that its counterpart went to *NK_DEV_VLINK_OFF* state, it shall finish/cancel all outstanding I/O operations, change its state to *NK_DEV_VLINK_OFF* and redo initialization.

The nk_vlink_lookup function searches for the first vlink with the *name* in the persistent device repository if parameter *plink* is equal to zero. Otherwise, *plink* must be a physical address returned by a previous call to nk_vlink_lookup. The next vlink with the *name*, starting from *plink* is returned in that case. If no vlink with the requested name is found, a null value is returned.

In other words, nk_vlink_lookup function can be called in a loop to find all devices for a given name.

If *name* is a NULL pointer, all the devices will be enumerated.

1.1.31.3.1 RETURN VALUES

In case of success, this primitive returns a physical address of the requested virtual link device descriptor. This physical address belongs to the global device repository and must be converted to a virtual one before usage through the nk_ptov primitive.

When no corresponding virtual link is found, a null value is returned.

1.1.31.4 SEE ALSO

nk_ptov

nk_vtop

1.1.32 NK_VTOP(3D)

1.1.32.1 NAME

nk_vtop — Return the physical address corresponding to a virtual address

1.1.32.2 SYNOPSIS

#include <nk/nkern.h>

NkPhAddr **nkops.nk_vtop**(void∗ *vaddr*);

1.1.32.3 DESCRIPTION

The Hypervisor supports processors with MMU. On such platforms, VMs may use different mappings for the physical memory. Because of that virtual addresses cannot be used as a pointers in the data structures accessed from different VMs. The physical addresses is used in this case. The NKDDI provides address translation functions nk_vtop, nk_ptov used for physical addresses belonging to the global persistent repository where virtual devices descriptors are stored.

The nk_vtop primitive translates a virtual memory address held in the *vaddr* parameter and belonging to the current virtual space into a physical address belonging to the global persistent device repository into a virtual address. Typically, virtual addresses returned by nk_ptov primitive can be translated into physical addresses through nk_vtop.

1.1.32.3.1 RETURN VALUES

This primitive returns a physical address belonging to the global persistent device repository and corresponding to the requested virtual address belonging to the current virtual space. In case of error, a null value is returned.

1.1.32.4 SEE ALSO

nk_ptov

1.1.33 NK_XIRQ_AFFINITY(3D)

1.1.33.1 NAME

nk_xirq_affinity — Set virtual CPU affinity for a given cross interrupt

1.1.33.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_xirq_affinity**(NkXirq *xirq*, NkCpuMask *cpus*);

1.1.33.3 DESCRIPTION

This primitive is invoked to set the virtual CPU affinity of a given cross interrupt.

The first *xirq* parameter is a valid cross interrupt value returned by the nk_pxirq_alloc primitive.

The second *cpus* parameter holds the associated virtual CPUs' identifier bitmask. A virtual CPU identifier affinity for a given cross interrupt is set when its corresponding bit is set in the bitmask according to the following statement: **1** << **CPU identifier**.

1.1.33.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.33.4 SEE ALSO

nk_pxirq_alloc

nk_xirq_attach

nk_xirq_detach

nk_xirq_mask

nk_xirq_ummask

nk_xirq_trigger

1.1.34 NK_XIRQ_ALLOC(3D)

1.1.34.1 NAME

nk_xirq_alloc — Allocate a contiguous range of free cross OS interrupts

1.1.34.2 SYNOPSIS

#include <nk/nkern.h>

NkXIrq **nkops.nk_xirq_alloc**(nku32_f *nb*);

1.1.34.3 RESTRICTIONS

This primitive is deprecated in the current version. The nk_pxirq_alloc must be used instead.

1.1.34.4 RETURN VALUES

This primitive always fails and returns 0.

1.1.35 NK_XIRQ_ATTACH(3D)

1.1.35.1 NAME

nk_xirq_attach — Attach a cross interrupt handler to a cross interrupt number

1.1.35.2 SYNOPSIS

#include <nk/nkern.h>

NkXIrqId **nkops.nk_xirq_attach**(NkXIrq *xirq*, NkXIrqHandler *hdl*, void∗ *cookie*);

with the following cross interrupt handler signature:

typedef void (∗**NkXIrqHandler**)(void∗ *cookie*, NkXIrq *xirq*)

1.1.35.3 DESCRIPTION

The first *xirq* parameter is a valid cross interrupt returned by the nk_pxirq_alloc primitive.

The second *hdl* parameter is the address of the cross interrupt handler to which the given cross interrupt is attached.

The last *cookie* parameter is an opaque value for the Hypervisor and is kept by the NKDDI and passed as the first parameter to the cross interrupt handler when it is called.

The nk_xirq_attach primitive attaches a cross interrupt handler *hdl* to the cross interrupt number *xirq*. A cross interrupt named *NK_XIRQ_SYSCONF* is dedicated for events concerning configuration or reconfiguration. This type of event is widely used by virtual drivers.

It is possible to attach multiple cross interrupt handlers to a same cross interrupt. All such handlers will be called in a loop.

1.1.35.3.1 RETURN VALUES

This function returns 0 in the case of failure. In case of success, this primitive returns a cross interrupt identifier. This return value must be passed to nk_xirq_detach primitive when a virtual driver is shutting down.

1.1.35.4 SEE ALSO

nk_pxirq_alloc

nk_xirq_affinity

nk_xirq_detach

nk_xirq_mask

nk_xirq_trigger

nk_xirq_unmask

1.1.36 NK_XIRQ_ATTACH_MASKED(3D)

1.1.36.1 NAME

nk_xirq_attach_masked — Attach a cross interrupt handler to a cross interrupt number

1.1.36.2 SYNOPSIS

#include <nk/nkern.h>

NkXIrqId **nkops.nk_xirq_attach_masked** (NkXIrq *xirq*, NkXIrqHandler *hdl*, void∗ *cookie*);

with the following cross interrupt handler signature:

typedef void (∗**NkXIrqHandler**) (void∗ *cookie*, NkXIrq *xirq*);

1.1.36.3 RESTRICTIONS

This primitive is deprecated in the current version. The nk_xirq_attach must be used instead.

1.1.36.4 RETURN VALUES

This primitive always fails, BUG() is issued on Linux.

1.1.37 NK_XIRQ_DETACH(3D)

1.1.37.1 NAME

nk_xirq_detach — Detach a cross interrupt handler

1.1.37.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_xirq_detach**(NkXIrqId *id*);

1.1.37.3 DESCRIPTION

The unique *id* parameter is a valid cross interrupt identifier returned by nk_xirq_attach primitive.

This primitive detaches the cross interrupt handler *id* previously attached. It will wait for the execution of current handler to terminate, if this handler happens to be executing on another CPU.

1.1.37.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.37.4 SEE ALSO

nk_pxirq_alloc

nk_xirq_attach

nk_xirq_mask

nk_xirq_ummask

nk_xirq_affinity

nk_xirq_trigger

1.1.38 NK_XIRQ_MASK(3D)

1.1.38.1 NAME

nk_xirq_mask — Mask a given cross interrupt

1.1.38.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_xirq_mask**(NkXIrq *xirq*);

1.1.38.3 DESCRIPTION

The unique *xirq* parameter is a valid cross interrupt returned by the nk_pxirq_alloc primitives.

This primitive is invoked to mask a cross interrupt given in *xirq* parameter.

The nk_xirq_mask function can be called from an interrupt handler.

1.1.38.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.38.4 SEE ALSO

nk_pxirq_alloc

nk_xirq_attach

nk_xirq_detach

nk_xirq_ummask

nk_xirq_affinity

nk_xirq_trigger

1.1.39 NK_XIRQ_TRIGGER(3D)

1.1.39.1 NAME

nk_xirq_trigger — Post a cross interrupt to a given guest operating system

1.1.39.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_xirq_trigger**(NkXIrq *xirq*, NkOsId *osid*);

1.1.39.3 DESCRIPTION

The first *xirq* parameter is a valid cross interrupt return by the nk_pxirq_alloc primitive.

The second *osid* parameter is a valid VM identifier.

This primitive posts cross interrupt *xirq* to the VM *osid*. This function is idempotent — if it is called multiple times before cross interrupt processing is started, the cross interrupt handlers attached to the cross interrupt *xirq* will be called once. As mentioned in the nk_xirq_attach manual page, a dedicated *NK_XIRQ_SYSCONF* is used for events concerning configuration or reconfiguration. This type of event is widely used by virtual drivers.

The nk_xirq_trigger function can be called from an interrupt handler.

1.1.39.3.1 RETURN VALUES

No relevant value is returned by this primitive.

1.1.39.4 SEE ALSO

nk_pxirq_alloc

nk_xirq_attach

nk_xirq_detach

nk_xirq_mask

nk_xirq_ummask

nk_xirq_affinity

1.1.40 NK_XIRQ_UNMASK(3D)

1.1.40.1 NAME

nk_xirq_unmask — Unmask a given cross interrupt

1.1.40.2 SYNOPSIS

#include <nk/nkern.h>

void **nkops.nk_xirq_unmask**(NkXIrq *xirq*);

1.1.40.3 DESCRIPTION

The unique *xirq* parameter is a valid cross interrupt returned by the nk_pxirq_alloc primitive.

This primitive is invoked to unmask a cross interrupt given in *xirq* parameter. If the unmasked cross interrupt was pending, the attached handler is immediately invoked.

The nk_xirq_unmask function can be called from an interrupt handler.

1.1.40.3.1 VALUES

No relevant value is returned by this primitive.

1.1.40.4 SEE ALSO

nk_pxirq_alloc

nk_xirq_attach

nk_xirq_detach

nk_xirq_mask

nk_xirq_affinity

nk_xirq_trigger

1.1.41 NK_SYSCONF_ATTACH(3D)

1.1.41.1 NAME

nk_sysconf_attach — Attach a handler function to the handshake xirq (legacy NK_XIRQ_SYSCONF) on the vlink end-point designated by "plink".

1.1.41.2 SYNOPSIS

#include <nk/nkern.h>

int **nk_sysconf_attach**(NkPhAddr *plink*, NkXIrqHandler *hdl*, void∗ *cookie*);

with the following cross interrupt handler signature:

typedef void (∗**NkXIrqHandler**)(void∗ *cookie*, NkXIrq *xirq*)

1.1.41.3 DESCRIPTION

The first *plink* parameter is a valid vlink end-point physical address returned by the nk_vlink_lookup() primitive.

The second *hdl* parameter is the address of the handler function to be attached.

The last *cookie* parameter is an opaque value for the Hypervisor and is kept by the NKDDI and passed as the first parameter to the handler function when it is called.

The nk_sysconf_attach() primitive attaches the handler function which is called each time a handshake xirq is signaled on the vlink end-point. The handshake xirq on a given vlink end-point can be signaled by either legacy nk_xirq_trigger(NK_XIRQ_SYSCONF) or modern nk_sysconf_post() / nk_sysconf_selfpost() primitives.

A single handler function can be attached to a given vlink end-point.

1.1.41.3.1 RETURN VALUES

This function returns 0 in the case of success, a negative value in the case of a failure.

1.1.41.4 SEE ALSO

nk_sysconf_detach

nk_sysconf_post

nk_sysconf_selfpost

1.1.42 NK_SYSCONF_DETACH(3D)

1.1.42.1 NAME

nk_sysconf_detach — Detach the handler function from the handshake xirq on the vlink end-point designated by "plink".

1.1.42.2 SYNOPSIS

#include <nk/nkern.h>

void **nk_sysconf_detach**(NkPhAddr *plink*);

1.1.42.3 DESCRIPTION

The *plink* parameter is a valid vlink end-point physical address returned by the nk_vlink_lookup() primitive.

The nk_sysconf_detach() primitive detaches the handler function from the handshake xirq which was previously attached with nk_sysconf_attach().

1.1.42.3.1 RETURN VALUES

None.

1.1.42.4 SEE ALSO

nk_sysconf_attach

nk_sysconf_post

nk_sysconf_selfpost

1.1.43 NK_SYSCONF_POST(3D)

1.1.43.1 NAME

nk_sysconf_post — signals a handshake xirq to the peer end-point of the vlink designated by the "plink".

1.1.43.2 SYNOPSIS

#include <nk/nkern.h>

void **nk_sysconf_post**(NkPhAddr *plink*);

1.1.43.3 DESCRIPTION

The *plink* parameter is a valid vlink physical address returned by the nk_vlink_lookup() primitive.

The nk_sysconf_post() primitive signals a handshake xirq to the peer end-point of the vlink designated by the "plink". It should be called only between nk_sysconf_attach() and nk_sysconf_detach() invocations, otherwise it does nothing. The method of the handshake xirq signaling depends on the peer virtual driver. If the peer virtual driver uses the nk_sysconf_attach() primitive then a per-vlink handshake xirq will be delivered, otherwise a legacy NK_XIRQ_SYSCONF will be delivered.

1.1.43.3.1 RETURN VALUES

None.

1.1.43.4 SEE ALSO

nk_sysconf_attach

nk_sysconf_detach

nk_sysconf_selfpost

1.1.44 NK_SYSCONF_SELFPOST(3D)

1.1.44.1 NAME

nk_sysconf_selfpost — signals a handshake xirq to the end-point of the vlink designated by the "plink".

1.1.44.2 SYNOPSIS

#include <nk/nkern.h>

void **nk_sysconf_selfpost**(NkPhAddr *plink*);

1.1.44.3 DESCRIPTION

The *plink* parameter is a valid vlink end-point physical address returned by the nk_pxirq_alloc() primitive.

The nk_sysconf_selfpost() primitive signals a handshake xirq to the end-point of the vlink designated by the "plink". It should be called only between nk_sysconf_attach() and nk_sysconf_detach() invocations, otherwise it does nothing.

1.1.44.3.1 RETURN VALUES

None.

1.1.44.4 SEE ALSO

nk_sysconf_attach

nk_sysconf_detach

nk_sysconf_selfpost

1.2 (4D) vdrivers

1.2.1 DOCILE_CLOCK(4D)

1.2.1.1 Cross References

Related Documents
Manual Page

1.2.1.2 NAME

docile_clock — Docile Clock Driver

1.2.1.3 SYNOPSIS

The docile clock driver implements a clock that accepts any rate change but doesn't perform any real hardware operation. This kind of clock is sometimes useful for virtualization, when you need to replace real clock nodes with a clock that does nothing in a device tree.

1.2.1.4 FEATURES

Docile clock driver can be enabled in the Linux kernel configuration:

Device Drivers -> VLX virtual device support -> VLX Docile Clock Driver

Under Linux, sources of clock signal are typically represented by nodes in the device tree. Those nodes are called **clock providers**. Device nodes that depend on those sources are called **clock consumers**. For a general overview of those concepts, please refer to `Linux clock device tree binding.`

Docile clock driver is a clock provider driver. A docile clock provider is defined by a node in the device tree that sets two properties: *compatible="vl,vclk-docile"* and a property *#clock-cells* specifying the dimension of the clock (i.e the number of integers required to index the clock from a clock consumer). Note that docile clock provider supports any clock dimension.

Below is a typical example of the definition of a docile clock provider and an associated clock consumer:

```
docile: docile-node {
    compatible = "vl,vclk-docile";
    #clock-cells = <2>;
};

consumer: consumer-node {
    compatible = "my-consumer-compatible";
    clocks = <&docile 12 1>;
};
```

1.2.1.5 NOTES

1.2.1.6 SEE ALSO

`Device tree bindings for clock providers and clock consumers.`

1.2.2 VAUDIO_BE(4D)

1.2.2.1 NAME

vaudio_be — Virtual AUDIO back-end driver interface

1.2.2.2 SYNOPSIS

#include <nk/nkern.h>
#include <vaudio.h>

1.2.2.3 FEATURES

The virtual audio back-end driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual Audio backend driver for VLX based Linux

It can be compiled as a loadable module or as an embedded driver.

Virtual audio devices must be declared in the platform device tree. The example below declares vaudio devices using the virtual link framework(see the **nk_vlink_lookup(3D)** manual page for more details).

```
&vm2_vdevs {

    vaudio_be: vaudio@be {          // vAUDIO backend for VM3
        compatible = "vaudio";      //
        server;                     // server end point
        #clone     = "auto";
        info       = "D32:64,d3:128,d8::0,d9::0,d10:128:0";
    };

};

&vm3_vdevs {

    vaudio@fe {                           // vAUDIO frontend
        peer-phandle = <&vaudio_be>;      // peer vLINK
        client;                           // client end point
    };

};
```

There is only one link in this example. The backend (server) side is managed by the backend vaudio driver running in the VM #2. The frontend (client) side is managed by the frontend driver running in VM #3.

The 'info' property can be used to define new values for playback and capture ring sizes on the backend side.

The syntax is: "[D[<p>][:<c>]]{[,d<d>:[<p>][:<c>]]}"

- "D" option defines new default values for all capture and/or playback ring sizes (kB)
- "d" option defines new values for capture and/or playback ring sizes (kB) of a device

1.2.2.4 DESCRIPTION

The virtual audio driver provides an API to frontend drivers running on the same or other VMs. The backend driver is responsible for accessing several physical audio devices through an underlying native driver. This API is also accessible by frontend drivers running on other VMs.

1.2.2.5 EXTENDED DESCRIPTION

The virtual audio device is an abstraction which enables a VM to access real audio devices managed by another VM. The vaudio backend driver is responsible for accessing the physical audio device via underlying native audio driver. It uses the NKDDI P2P communication link to provide communications and synchronization between frontend and backend drivers.

The backend driver creates a virtual audio driver instance and then is entirely driven by the frontend driver. To achieve that goal, the backend driver receives control events from frontend drivers through the Hypervisor P2P communication link.

The virtual audio backend driver exports *NK_VAUDIO_DEV_MAX* virtual audio devices and a single virtual audio mixer in the Hypervisor repository (shared persistent memory).

Each virtual audio device exports *NK_VAUDIO_STREAM_MAX* streams.

The virtual audio mixer object has the following definition:

```
    /*!
     * Values for the mix_cmd field of the NkEventMixer event.
     */
#define NK_VAUDIO_MIXER_INFO        1
#define NK_VAUDIO_MIXER_GET         2
#define NK_VAUDIO_MIXER_PUT         3

    /*!
     * Values for the mix_info.type field of the NkEventMixer event.
     */
#define NK_CTL_ELEM_TYPE_NONE       0
#define NK_CTL_ELEM_TYPE_BOOLEAN    1
#define NK_CTL_ELEM_TYPE_INTEGER    2
#define NK_CTL_ELEM_TYPE_ENUMERATED 3
#define NK_CTL_ELEM_TYPE_LAST       NK_CTL_ELEM_TYPE_ENUMERATED

    /*!
     * NK_VAUDIO_STREAM_MIXER event parameters.
     */
#define NK_VAUDIO_MIXER_MAX 128

typedef struct {
    nku8_f   name[64];
    nku32_f  type;
    nku32_f  count;
    union {
    struct {
        nku32_f min;
        nku32_f max;
        nku32_f step;
    } integer;
    struct {
        nku32_f items;
        nku32_f item;
        nku8_f  name[64];
    } enumerated;
    } value;
    nku8_f  reserved[64];
} NkCtlElemInfo;

typedef struct {
    union {
        struct {
        nku32_f value[8];
    } integer;
    struct {
        nku32_f item[8];
    } enumerated;
    } value;
} NkCtlElemValue;
```

```
typedef struct {
    nku32_f        mix_cmd;
    nku32_f        mix_idx;
    nku32_f        mix_type;
    NkCtlElemInfo  mix_info;
    NkCtlElemValue mix_val;
} NkEventMixer;
```

A virtual audio stream object has the following definition:

NkDevRing object used to exchange data between frontend and backend drivers.

A pair of *NkVaudioHw* objects for play back and capture (that is, record) modes.

NkVaudioCtrl object containing control information exchanged between backend and frontend drivers.

The first *NkDevRing* object is exported by the Hypervisor and is specific data for communication ring device:

```
typedef struct NkDevRing {
    NkOsId      pid;            /* Producer OS ID */
    nku32_f     type;           /* Ring type */
    nku32_f     cxirq;          /* Consumer XIRQ */
    nku32_f     pxirq;          /* Producer XIRQ */
    nku32_f     dsize;          /* Ring descriptor size */
    nku32_f     imask;          /* Ring index mask */
    nku32_f     iresp;          /* Consumer response ring index */
    nku32_f     ireq;           /* Producer request ring index */
    NkPhAddr    base;           /* Ring physical base address */
} NkDevRing;

#define   RING_DESC_NB   64

typedef struct NkRingDesc {
    nku32_f     status;
    nku32_f bufsize;
    NkPhAddr    bufaddr;
} NkRingDesc;
```

The ring is provided by a consumer VM, usually the VM running the backend driver, for a given producer VM, usually the VM running the frontend driver.

The **pid** holds the consumer VM identifier. If the ring consumer is unknown, this field is set to zero. In that case, this field can be updated by the producer at connection time.

The **type** field contains the ring type on four ASCII characters (that is, "DISK" for a remote disk ring and so forth).

The **cxirq** holds the persistent cross interrupts for consumer (backend driver) notifications (see also **nk_pxirq_↩ alloc(3D)** manual page for further details)./n The **pxirq** holds the persistent cross-interrupts for producer (frontend driver) notifications.

The **dsize** is set to the size of a *NkRingDesc* object by the backend driver (see **vaudio_create** primitive).

The **imask** field is set to **RING_DESC_NB — 1**. This field is the ring index mask and must hold all the relevant indexes corresponding to the number of rings (that is, this number must always be a power of 2). There are **RIN↩ G_DESC_NB** ring buffers for data exchange between the frontend and the backend drivers.

Both **iresp** and **ireq** are set to zero. These fields are respectively the first available index for a consumer response, and the first available index for a producer request. These fields are managed as free run counters. Typically, the consumer and producer send a cross interrupt through respectively **cxirq** and **pxirq**, then **iresp** and **ireq** are incremented. A logic *AND* operation is always done with **imask** and the related counter in order to get the offset of the first available ring.

The **base** holds the physical base address of the ring (see **nkdevops.h** for further details).

The second *NkVaudioHw* is defined in **vaudio.h** file and has the following layout:

```
    /*! NkVaudioHwPcm : configuration for the PCM stream type.
     *   formats       : PCM formats supported
     *   rates         : sample rates supported
     *   rate_min      : minimal rate in HZ
     *   rate_min      : maximal rate in HZ
     *   channels_min : minimal number of channels
     *   channels_max : maximal number of channels
     */
typedef struct {
    nku64_f formats;
    nku32_f rates;
    nku32_f rate_min;
    nku32_f rate_max;
    nku32_f channels_min;
    nku32_f channels_max;
    nku32_f buffer_bytes_max;
    nku32_f period_bytes_min;
    nku32_f period_bytes_max;
    nku32_f periods_min;
    nku32_f periods_max;
    nku32_f fifo_size;
    nku32_f pad64;
} NkVaudioHwPcm;

    /*!
     * Values for the stream_cap field of the NkVaudioHw configuration.
     */
#define HW_CAP_PCM    0x00000001
#define HW_CAP_DMA    0x80000000

    /*! NkVaudioHw : hardware configuration for the audio device.
     *    stream_cap : supported stream types
     *    pcm        : PCM configuration
     */
typedef struct {
    nku32_f                  stream_cap;
    nku32_f                  pad64;
    NkVaudioHwPcm            pcm;
} NkVaudioHw;
```

This object is initialized by the the backend driver at initialization time.

The last *NkVaudioCtrl* exported object is defined below and has the following layout:

```
    /* vAUDIO commands */
#define NK_VAUDIO_COMMAND_NONE       0x00000000
#define NK_VAUDIO_COMMAND_OPEN       0x00000001
#define NK_VAUDIO_COMMAND_CLOSE      0x00000002
#define NK_VAUDIO_COMMAND_START      0x00000004
#define NK_VAUDIO_COMMAND_STOP       0x00000008
#define NK_VAUDIO_COMMAND_SET_RATE   0x00000010
#define NK_VAUDIO_COMMAND_MIXER      0x00000020

    /*!
     * Error codes for the vaudio_ring_put and vaudio_event_ack
     * status parameters.
     */
```

```
typedef enum {
    NK_VAUDIO_STATUS_OK    = 0,
    NK_VAUDIO_STATUS_ERROR = -1
} vaudio_status_t;

    /*!
     * Values for the session_type field of the NkEventOpen event.
     */
#define NK_VAUDIO_SS_TYPE_INVAL      0
#define NK_VAUDIO_SS_TYPE_PLAYBACK   1
#define NK_VAUDIO_SS_TYPE_CAPTURE    2

    /*!
     * Values for the stream_type field of the NkEventOpen event.
     */
#define NK_VAUDIO_ST_TYPE_INVAL      0
#define NK_VAUDIO_ST_TYPE_PCM        1

    /*!
     * Values for the format field of the NkEventSetRate event.
     */
typedef enum {
    HW_PCM_FORMAT_S8 = 0,
    HW_PCM_FORMAT_U8,
    HW_PCM_FORMAT_S16_LE,
    HW_PCM_FORMAT_S16_BE,
    HW_PCM_FORMAT_U16_LE,
    HW_PCM_FORMAT_U16_BE,
    HW_PCM_FORMAT_S24_LE,
    HW_PCM_FORMAT_S24_BE,
    HW_PCM_FORMAT_U24_LE,
    HW_PCM_FORMAT_U24_BE,
    HW_PCM_FORMAT_S32_LE,
    HW_PCM_FORMAT_S32_BE,
    HW_PCM_FORMAT_U32_LE,
    HW_PCM_FORMAT_U32_BE,
    HW_PCM_FORMAT_MAX
} vaudio_format_t;

    /*!
     * Virtual audio control definition.
     */
typedef struct {
    nku32_f    cxirq;
    nku32_f    pxirq;
    nku32_f    command;
    nku32_f    status;

    nku8_f     session_type;
    nku8_f     stream_type;
    nku8_f         channels;
    nku8_f         format;
    nku32_f    rate;
    nku32_f    period;
    nku32_f    periods;
} NkVaudioCtrl;
```

This data structure is used to exchange the current status of the virtual audio device between front-end and back-end drivers.

A *NkRingDesc* object gives the size and physical address of a data buffer. When the audio device is in playback mode, the backend driver gets a buffer from the ring plays data in the buffer and puts back the buffer in the ring with a status as the result of the operation. When the audio device is in record mode (that is, capture mode), the backend driver gets a buffer from the ring fills the buffer and puts back the buffer in the ring.

When the backend driver has complete all data object initialization, the virtual link handshake protocol is started to enable communications between all peers counterpart (that is, frontend drivers). The **nk_vlink_lookup(3D)** manual page can be consulted for further details about this handshake protocol.

The virtual audio backend driver API allows a frontend driver to perform the following operations:

Create a virtual audio device

Get control events

Acknowledge control events (callback)

Get data buffers from the ring

Put data buffers to the ring

According to the host OS, virtual audio backend driver accesses the native audio device through basic operations depending of the services exported by this native driver.

1.2.2.6 CREATING A VIRTUAL AUDIO DEVICE

1.2.2.6.1 SYNOPSIS

Prior to using a virtual audio device, it must be created. This is done by invoking the vaudio_create() primitive.

#include <vaudio.h>

NkVaudio **vaudio_create** (NkDevVlink∗ *vlink*, NkVaudioEventHandler *hdl*, void∗ *cookie*, NkVaudioHw∗ *hw_conf*);

1.2.2.6.2 DESCRIPTION

This routine creates a virtual audio device.

1.2.2.6.3 PARAMETERS

The *vlink* parameter specifies the virtual link used to communicate with the frontend.

The parameter *hdl* specifies the event handler. The event handler is invoked in interrupt handling context. So an event handler should not use API which are not allowed by the underlying VM within interrupt handlers.

The parameter *cookie* will be passed as a parameter to the event handler.

The parameter *hw_conf* specifies the audio hardware configuration.

The list of *NkVaudioEventHandler* possible events is given below:

```
/*
 * Here are the virtual AUDIO events\&.
 */
typedef enum {
    /* Indicate that remote site has opened the device */
    NK_VAUDIO_STREAM_OPEN              = 0x0,

    /* Indicate that remote site has closed the device */
    NK_VAUDIO_STREAM_CLOSE             = 0x1,

    /* Start the audio stream */
    NK_VAUDIO_STREAM_START             = 0x2,

    /* Stop the audio stream */
    NK_VAUDIO_STREAM_STOP              = 0x3,

    /* Set the rate of the audio stream */
    NK_VAUDIO_STREAM_SET_RATE          = 0x4,

    /* Indicate that a data buffer is available */
    NK_VAUDIO_STREAM_DATA              = 0x5,

    /* Get/set stream volume control */
    NK_VAUDIO_STREAM_MIXER             = 0x6,

    NK_VAUDIO_EVENT_MAX        = 0x7

} NkVaudioEvent;

    /*!
     * NK_VAUDIO_STREAM_OPEN event parameters:
     *   session_type : playback or capture
     *   stream_type  : PCM only
     */
typedef struct {
    nku8_f    session_type;
    nku8_f    stream_type;
} NkEventOpen;

typedef NkEventOpen NkEventClose;

    /*!
     * NK_VAUDIO_STREAM_SET_RATE event parameters:
     *   channels : number of channels
     *   format   : endianess and size of samples
     *   rate     : rate in HZ
     *   period   : period size in bytes
     *   periods  : number of periods in the ring buffer
     *   dma_paddr: dma physical address of the ring buffer
     *   dma_vaddr: virtual address of the ring buffer
     * This is not a shared memory object between backend and frontend.
     */
typedef struct {
    nku8_f     channels;
    nku8_f     format;
    nku32_f    rate;
    nku32_f    period;
    nku32_f    periods;
    NkPhAddr   dma_paddr;
    void*      dma_vaddr;
} NkEventSetRate;

/*
 * NK_VAUDIO_STREAM_MIXER event parameters:
 */
typedef struct {
    nku32_f        mix_cmd;    /* Mixer commands: NK_VAUDIO_MIXER_XXX */
    nku32_f        mix_idx;
    nku32_f        mix_type;
    NkCtlElemInfo  mix_info;
    NkCtlElemValue mix_val;
} NkEventMixer;
```

The list of *NkVaudioHwPcm* formats and rates are defined below:

```
    /*!
     * Values for the formats field of the NkVaudioHwPcm configuration.
     */
#define HW_PCM_FMTBIT_S8                (1ULL << HW_PCM_FORMAT_S8)
#define HW_PCM_FMTBIT_U8                (1ULL << HW_PCM_FORMAT_U8)
#define HW_PCM_FMTBIT_S16_LE        (1ULL << HW_PCM_FORMAT_S16_LE)
#define HW_PCM_FMTBIT_S16_BE        (1ULL << HW_PCM_FORMAT_S16_BE)
#define HW_PCM_FMTBIT_U16_LE        (1ULL << HW_PCM_FORMAT_U16_LE)
#define HW_PCM_FMTBIT_U16_BE        (1ULL << HW_PCM_FORMAT_U16_BE)
#define HW_PCM_FMTBIT_S24_LE        (1ULL << HW_PCM_FORMAT_S24_LE)
#define HW_PCM_FMTBIT_S24_BE        (1ULL << HW_PCM_FORMAT_S24_BE)
#define HW_PCM_FMTBIT_U24_LE        (1ULL << HW_PCM_FORMAT_U24_LE)
#define HW_PCM_FMTBIT_U24_BE        (1ULL << HW_PCM_FORMAT_U24_BE)
#define HW_PCM_FMTBIT_S32_LE        (1ULL << HW_PCM_FORMAT_S32_LE)
#define HW_PCM_FMTBIT_S32_BE        (1ULL << HW_PCM_FORMAT_S32_BE)
#define HW_PCM_FMTBIT_U32_LE        (1ULL << HW_PCM_FORMAT_U32_LE)
#define HW_PCM_FMTBIT_U32_BE        (1ULL << HW_PCM_FORMAT_U32_BE)
#define HW_PCM_FMTBIT_ALL           ((1ULL << HW_PCM_FORMAT_MAX) - 1)

    /*!
     * Values for the rates field of the NkVaudioHwPcm configuration.
     */
#define HW_PCM_RATE_5512                (1<<0)      /* 5512Hz */
#define HW_PCM_RATE_8000                (1<<1)      /* 8000Hz */
#define HW_PCM_RATE_11025       (1<<2)      /* 11025Hz */
#define HW_PCM_RATE_16000       (1<<3)      /* 16000Hz */
#define HW_PCM_RATE_22050       (1<<4)      /* 22050Hz */
#define HW_PCM_RATE_32000       (1<<5)      /* 32000Hz */
#define HW_PCM_RATE_44100       (1<<6)      /* 44100Hz */
#define HW_PCM_RATE_48000       (1<<7)      /* 48000Hz */
#define HW_PCM_RATE_64000       (1<<8)      /* 64000Hz */
#define HW_PCM_RATE_88200       (1<<9)      /* 88200Hz */
#define HW_PCM_RATE_96000       (1<<10)     /* 96000Hz */
#define HW_PCM_RATE_176400      (1<<11)     /* 176400Hz */
#define HW_PCM_RATE_192000      (1<<12)     /* 192000Hz */
#define HW_PCM_RATE_ALL             ((1<<13) - 1)
```

1.2.2.6.4 RETURN VALUES

This routine returns a handle on the created virtual audio device in case of success or 0 if an error has occurred. This handle is a pointer on the following object:

```
struct NkVaudio {
    NkDevVlink*          vlink;
    NkVaudioEventHandler hdl;
    void*                cookie;
    NkStream             stream[NK_VAUDIO_DEV_MAX][NK_VAUDIO_STREAM_MAX];
    vaudio_shmem_t       shmem;
};

struct NkStream {
    nku32_f                req;
    NkVaudio*              vaudio;
    void*                  cookie;
    vaudio_stream_shmem_t* shmem;
};
```

1.2.2.7 ACKNOWLEDGING AN EVENT

1.2.2.7.1 SYNOPSIS

When an event has been received and processed by the backend driver, an acknowledgment must be returned to the frontend driver. This should be done by invoking the vaudio_event_ack() function invoked by the internal *NkVaudioEventHandler* interrupt handler (*vaudio_HISR*).

#include <vaudio.h>

void **vaudio_event_ack** (NkVaudio *vaudio*, NkStream *stream*, NkVaudioEvent *event*, void∗ *params*, nku32_f *status*);

typedef void (*NkVaudioEventHandler) (void* stream_cookie, NkVaudioEvent event, void∗ params, void∗ vaudio_↩ cookie);

The *NkVaudioEventHandler* callback **vaudio_HISR** is invoked by the virtual audio interrupt handler connect by **nk_xirq_attach(3D)** primitive.

1.2.2.7.2 DESCRIPTION

When an event has been triggered by the frontend driver, an interrupt occurs and the backend event handler is invoked. Usually the event cannot be managed in the context of the event handler because it is an interrupt context. So the event handler should return as soon as possible after having saved event parameters and awoken a thread which will manage the event. After the operation has been completed, the vaudio_event_ack() must be invoked returning the appropriate return code in the *status* parameter.

1.2.2.7.3 PARAMETERS

The *vaudio* parameter is a valid virtual audio handle. In other words, this handle is a pointer to a valid *NkVaudio* object (see previous section: CREATING A VIRTUAL AUDIO DEVICE for further details about this object.

The second argument *stream* is a pointer to a valid *NkStream* object.

The third argument *event* specifies the event to be acknowledged.

The fourth argument *params* is a pointer to a *NkEventMixer* object already mentioned above and corresponding to get a mixer value.

The last argument is the *status* parameter which is set to *NK_VAUDIO_STATUS_OK* in case of success or set to *NK_VAUDIO_STATUS_ERROR* otherwise.

1.2.2.7.4 RETURN VALUES

This call back handler itself returns no relevant returned code.

1.2.2.8 GETTING A DATA BUFFER

1.2.2.8.1 SYNOPSIS

The vaudio_ring_get() function is used to get the first available data buffer from the virtual audio ring.

#include <vaudio.h>

int **vaudio_ring_get** (NkStream *stream*, NkPhAddr∗ *addr*, nku32_f∗ *size*);

1.2.2.8.2 DESCRIPTION

If the opened stream is in playback mode, the buffer contains data to be played by the backend driver. If the opened stream is in capture mode, the buffer will be filled by the backend driver.

1.2.2.8.3 PARAMETERS

The first argument *stream* is a valid virtual audio stream handle.

The second argument *addr* refers a pointer where the physical address of the buffer will be put.

The last argument *size* holds a pointer where the size of the buffer will be put.

1.2.2.8.4 RETURN VALUES

This routine returns 1 if a data buffer is available and 0 otherwise.

1.2.2.9 PUTTING BACK A DATA BUFFER

1.2.2.9.1 SYNOPSIS

The vaudio_ring_put() function is used to put back the oldest used data buffer to the virtual audio ring.

#include <vaudio.h>

void **vaudio_ring_put** (NkStream *stream*, nku32_f *status*);

1.2.2.9.2 DESCRIPTION

If the opened stream is in capture mode, the buffer contains data captured by the backend driver. If the opened stream is in playback mode, the buffer has been played by the backend driver.

1.2.2.9.3 PARAMETERS

The first argument *stream* is a valid virtual audio stream handle.

The last argument *status* holds the returned error code.

1.2.2.9.4 RETURN VALUES

This primitive returns no relevant returned code.

1.2.2.10 SEE ALSO

vaudio_fe

nk_id_get

nk_pdev_alloc

nk_ptov

nk_vtop

vlink_lookup

nk_xirq_trigger

nk_xirq_attach

nk_xirq_detach

1.2.3 VAUDIO_FE(4D)

1.2.3.1 NAME

vaudio_fe — Virtual AUDIO frontend driver interface

1.2.3.2 SYNOPSIS

#include <nk/nkern.h>
#include <vaudio.h>

The virtual audio frontend driver is based upon ALSA (Advanced Linux Sound Architecture).

1.2.3.3 FEATURES

The virtual audio frontend driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual Audio frontend driver for VLX based Linux

It can be compiled as a loadable module or as an embedded driver.

Virtual devices are declared in the platform device tree for each VM. An example is given in the vaudio_be manual page.

1.2.3.4 VIRTUAL LINKS

The virtual audio frontend driver is based on the virtual link framework described in **nk_vlink_lookup(3D)** manual page.

1.2.3.5 DESCRIPTION

The virtual audio device is an abstraction which enables a VM to access real audio devices managed by another VM. It exports standard Linux API for the audio devices. A virtual audio frontend driver running on a VM invokes the backend exported API to access real audio devices.

1.2.3.6 EXTENDED DESCRIPTION

The virtual audio front-driver uses the NKDDI to provide communications and synchronization between frontend and backend drivers.

The backend driver creates a virtual audio driver instance and then is entirely driven by the frontend driver. To achieve that goal, the back end driver receives control events from the virtual audio device.

The virtual audio device exports a ring of descriptors to enable data communication between frontend and backend drivers. The exported data structures are described in details in the vaudio_be manual page. A descriptor gives the size and physical address of a data buffer. When the audio device is in playback mode, the backend driver gets a buffer from the ring, plays data in the buffer and puts back the buffer in the ring with a status as the result of the operation. When the audio device is in record mode, the backend driver gets a buffer from the ring, fills the buffer and puts back the buffer in the ring.

When the frontend driver is initialized, the virtual link protocol handshake is started up to establish a communication link with its peer counterpart (that is, the backend driver). The **nk_vlink_lookup(3D)** manual page can be consulted for further details about this handshake protocol.

The virtual audio backend driver API allows a frontend driver to perform the following operations:

- Create a virtual audio driver instance
- Get control events
- Acknowledge control events (callback)
- Get data buffers from the data ring
- Put data buffers to the data ring

This API is described in the **vaudio_be(4D)** manual page.

In addition, the virtual audio frontend driver exports a Pulse Code Modulation (PCM) services to the ALSA library. According to the following *snd_pcm_ops*, tables of functions and *snd_pcm_hardware* objects containing audio hardware features (see **sound/pcm.h** and **sound/asound.h** files for further details), the following data structures are managed by the virtual audio frontend driver:

```
typedef struct snd_pcm_ops {
    int (*open)(snd_pcm_substream* substream);
    int (*close)(snd_pcm_substream* substream);
    int (*ioctl)(snd_pcm_substream* substream, unsigned int cmd, void *arg);
    int (*hw_params)(snd_pcm_substream* substream, snd_pcm_hw_params *params);
    int (*hw_free)(snd_pcm_substream* substream);
    int (*prepare)(snd_pcm_substream* substream);
    int (*trigger)(snd_pcm_substream* substream, int cmd);
    snd_pcm_uframes_t (*pointer)(snd_pcm_substream* substream);
    int (*get_time_info)(struct snd_pcm_substream *substream,
             struct timespec *system_ts, struct timespec *audio_ts,
             struct snd_pcm_audio_tstamp_config *audio_tstamp_config,
             struct snd_pcm_audio_tstamp_report *audio_tstamp_report);
    int (*copy)(snd_pcm_substream* substream, int channel,
                 snd_pcm_uframes_t pos, void __user *buf, snd_pcm_uframes_t count);
```

```
    int (*silence)(snd_pcm_substream* substream, int channel,
                   snd_pcm_uframes_t pos, snd_pcm_uframes_t count);
    page* (*page)(snd_pcm_substream* substream, unsigned long offset);
    int (*mmap)(snd_pcm_substream* substream, vm_area_struct* vma);
    int (*ack)(snd_pcm_substream* substream);
} snd_pcm_ops;

enum {
    SNDRV_PCM_STREAM_PLAYBACK = 0,
    SNDRV_PCM_STREAM_CAPTURE,
    SNDRV_PCM_STREAM_LAST = SNDRV_PCM_STREAM_CAPTURE,
};

    /* Hardware supports mmap */
@#define SNDRV_PCM_INFO_MMAP                        0x00000001
    /* Period data are valid during transfer */
@#define SNDRV_PCM_INFO_MMAP_VALID                  0x00000002
    /* Double buffering needed for PCM start/stop */
@#define SNDRV_PCM_INFO_DOUBLE                      0x00000004
    /* Double buffering */
@#define SNDRV_PCM_INFO_BATCH                       0x00000010
    /* Channels are interleaved */
@#define SNDRV_PCM_INFO_INTERLEAVED                 0x00000100
    /* Channels are not interleaved */
@#define SNDRV_PCM_INFO_NONINTERLEAVED              0x00000200
    /* Complex frame organization (mmap only) */
@#define SNDRV_PCM_INFO_COMPLEX                     0x00000400
    /* Hardware transfer block of samples */
@#define SNDRV_PCM_INFO_BLOCK_TRANSFER              0x00010000
    /* Hardware supports ADC (capture) overrange detection */
@#define SNDRV_PCM_INFO_OVERRANGE                   0x00020000
    /* Hardware supports stream resume after suspend */
@#define SNDRV_PCM_INFO_RESUME                      0x00040000
    /* Pause ioctl is supported */
@#define SNDRV_PCM_INFO_PAUSE                       0x00080000
    /* Only half duplex */
@#define SNDRV_PCM_INFO_HALF_DUPLEX                 0x00100000
    /* Playback and capture stream are somewhat correlated */
@#define SNDRV_PCM_INFO_JOINT_DUPLEX                0x00200000
    /* PCM support some kind of sync go */
@#define SNDRV_PCM_INFO_SYNC_START                  0x00400000
    /* period wakeup can be disabled */
@#define SNDRV_PCM_INFO_NO_PERIOD_WAKEUP            0x00800000
    /* (Deprecated)has audio wall clock for audio/system time sync */
@#define SNDRV_PCM_INFO_HAS_WALL_CLOCK              0x01000000
    /* (Deprecated)has audio wall clock for audio/system time sync */
@#define SNDRV_PCM_INFO_HAS_LINK_ATIME              0x01000000
    /* report absolute hardware link audio time, not reset on startup */
@#define SNDRV_PCM_INFO_HAS_LINK_ABSOLUTE_ATIME     0x02000000
    /* report estimated link audio time */
@#define SNDRV_PCM_INFO_HAS_LINK_ESTIMATED_ATIME    0x04000000
    /* report synchronized audio/system time */
@#define SNDRV_PCM_INFO_HAS_LINK_SYNCHRONIZED_ATIME 0x08000000
    /* internal kernel flag - trigger in drain */
@#define SNDRV_PCM_INFO_DRAIN_TRIGGER               0x40000000
    /* internal kernel flag - FIFO size is in frames */
@#define SNDRV_PCM_INFO_FIFO_IN_FRAMES              0x80000000

typedef struct snd_pcm_hardware {
    unsigned int info;              /* SNDRV_PCM_INFO_* */
    u64          formats;           /* SNDRV_PCM_FMTBIT_* */
    unsigned int rates;             /* SNDRV_PCM_RATE_* */
    unsigned int rate_min;          /* min rate */
    unsigned int rate_max;          /* max rate */
    unsigned int channels_min;      /* min channels */
    unsigned int channels_max;      /* max channels */
    size_t       buffer_bytes_max;  /* max buffer size */
    size_t       period_bytes_min;  /* min period size */
    size_t       period_bytes_max;  /* max period size */
    unsigned int periods_min;       /* min # of periods */
    unsigned int periods_max;       /* max # of periods */
    size_t       fifo_size;         /* fifo size in bytes */
} snd_pcm_hardware;
```

All the pointers to function of *snd_pcm_ops* table are filled with virtual audio routines except the following fields↩ :**get_time_info**, **copy**, **silence**, **page** and **ack** which are not used by the virtual audio frontend driver.

vaudio_snd_open : Open an audio stream.

vaudio_snd_close : Close an audio stream.

vaudio_snd_ioctl : Audio I/O controls.

vaudio_snd_hw_params : Allocate a DMA buffer.

vaudio_snd_hw_free : Release a DMA buffer.

vaudio_snd_prepare : Setup an audio command.

vaudio_snd_trigger : Start or stop playback or capture an audio stream.

vaudio_snd_pointer : Return the processed number of frames.

vaudio_snd_mmap : Remap kernel memory to userspace .

The virtual audio frontend driver exports *NK_VAUDIO_DEV_MAX* virtual audio devices, each device comprising *NK_VAUDIO_STREAM_MAX* streams. One *snd_pcm_ops* object is declared in the virtual audio frontend driver for each stream. A similar *snd_pcm_hardware* object is also declared for the same purpose.

A *snd_pcm_hardware* object is statically initialized with the following values:

info is or-ed with the following flags: **SNDRV_PCM_INFO_INTERLEAVED**, **SNDRV_PCM_INFO_BLOCK_TRA↩ NSFER**, **SNDRV_PCM_INFO_MMAP**, and **SNDRV_PCM_INFO_MMAP_VALID**

All other fields **formats**, **rates**, **rate_min**, **rate_max**, **channels_min**, and **channels_max** are dynamically initialized by the virtual audio frontend driver.

1.2.3.7 OPENING AN AUDIO STREAM

1.2.3.7.1 SYNOPSIS

Prior to send audio commands, an audio stream must be opened through the **vaudio_snd_open** primitive.

#include <nk/nkern.h>
#include <vaudio.h>

int **vaudio_snd_open** (snd_pcm_substream∗ *substream*);

1.2.3.7.2 DESCRIPTION

This primitive updates the current *substream* object according to the mode: playback or capture (that is, stream identifier located in **substream->pstr->stream**, see **sound/pcm.h** for further details). The underlying object *NkVaudio* (see the **vaudio_be(4D)** manual page in section CREATING A VIRTUAL DEVICE for further details) is updated accordingly, the current audio command is set to *NK_VAUDIO_COMMAND_OPEN* and the current stream type is set to *NK_VAUDIO_ST_TYPE_PCM*. Finally, a cross interrupt is triggered to the backend driver. The backend driver processes this event through **vaudio_intr_ctrl** primitive and a returned code is set in the **status** field of a *NkVaudioCtrl* object.

1.2.3.7.3 PARAMETERS

The single argument is a valid pointer to a *struct snd_pcm_substream* object (see **sound/pcm.h** file for further details).

1.2.3.7.4 RETURN VALUES

The returned code is gotten after invoking the frontend driver, as explained below. In case of success, *NK_VAU↩ DIO_STATUS_OK* is returned, or *NK_VAUDIO_STATUS_ERROR* in case of error.

1.2.3.8 CLOSING AN AUDIO STREAM

1.2.3.8.1 SYNOPSIS

An audio stream is closed when **vaudio_snd_close** is invoked.

#include <nk/nkern.h>
#include <vaudio.h>

int **vaudio_snd_close** (snd_pcm_substream∗ *substream*);

1.2.3.8.2 DESCRIPTION

This primitive sets the current session stream to an invalid session (*NK_VAUDIO_SS_TYPE_INVAL*), the current stream type to an invalid type (*NK_VAUDIO_ST_TYPE_INVAL*), and the current command to *NK_VAUDIO_CO↩ MMAND_CLOSE*. Finally, the backend driver is invoked through a cross interrupt and handled by **vaudio_intr_ctrl**.

1.2.3.8.3 PARAMETERS

The single argument is a valid pointer to a *struct snd_pcm_substream* object (see **sound/pcm.h** file for further details).

1.2.3.8.4 RETURN VALUES

In case of success, *NK_VAUDIO_STATUS_OK* is returned, or *NK_VAUDIO_STATUS_ERROR* in case of error.

1.2.3.9 SUPPORTED I/O CONTROL FOR AN AUDIO DEVICE

1.2.3.9.1 SYNOPSIS

I/O controls on virtual audio devices are provided by **vaudio_snd_ioctl**.

#include <nk/nkern.h>
#include <vaudio.h>

int **vaudio_snd_ioctl** (snd_pcm_substream∗ *substream*, unsigned int *cmd*, void∗ *arg*);

1.2.3.9.2 DESCRIPTION

This primitive is based on the kernel Linux **ioctl**, see **sound/core/pcm_lib.c** file for further details.

1.2.3.9.3 PARAMETERS

The first argument is a valid pointer to a *struct snd_pcm_substream* object. The second *cmd* argument holds a valid I/O controls among the following values: *SNDRV_PCM_IOCTL1_INFO*, *SNDRV_PCM_IOCTL1_RESET*, and *SN↩ DRV_PCM_IOCTL1_CHANNEL_INFO*. The first one is not implemented, the second is invoked to reset the current audio stream and the last one is used to set up channel information. For this last case, the third argument must be a valid pointer to a *struct snd_pcm_channel_info* object (see **sound/core/pcm_lib.c**, **sound/core/pcm_misc.c**, and **sound/asound.h** files for further details).

1.2.3.9.4 RETURN VALUES

In case of success, 0 is returned. If the I/O control value is not supported *ENXIO* is returned, if the current stream format parameters are incorrect *EINVAL* is returned (see **sound/core/pcm_lib.c**, **sound/core/pcm_misc.c** files for further details).

1.2.3.10 ALLOCATING A DMA BUFFER

1.2.3.10.1 SYNOPSIS

A DMA buffer is allocated by **vaudio_snd_hw_params** primitive.

#include <nk/nkern.h>
#include <vaudio.h>

int **vaudio_snd_hw_params** (snd_pcm_substream∗ *substream*, snd_pcm_hw_params∗ *hw_params*);

1.2.3.10.2 DESCRIPTION

This primitive is based on the Linux kernel primitive **snd_pcm_lib_malloc_pages**, see **sound/core/pcm_↩ memory.c** file for further details.

1.2.3.10.3 PARAMETERS

The first argument is a valid pointer to a *struct snd_pcm_substream* object. The second parameter is a valid pointer to a *snd_pcm_hw_params* object (see **sound/pcm.h** and **sound/asound.h** files for further details).

1.2.3.10.4 RETURN VALUES

This primitive always returns 1.

1.2.3.11 RELEASING A DMA BUFFER

1.2.3.11.1 SYNOPSIS

#include <nk/nkern.h>
#include <vaudio.h>

A DMA buffer is released by **vaudio_snd_hw_free** primitive.

int **vaudio_snd_hw_free** (snd_pcm_substream∗ *substream*);

1.2.3.11.2 DESCRIPTION

This primitive is based on the Linux kernel primitive **snd_pcm_lib_free_pages**, see **sound/core/pcm_memory.c** file for further details.

1.2.3.11.3 PARAMETERS

The single argument is a valid pointer to a *struct snd_pcm_substream* object (see **sound/pcm.h** file for further details).

1.2.3.11.4 RETURN VALUES

This primitive always returns 0.

1.2.3.12 SETTING UP AN AUDIO COMMAND

1.2.3.12.1 SYNOPSIS

A new rate on an audio stream is setup by **vaudio_snd_prepare** primitive.

#include <nk/nkern.h>
#include <vaudio.h>

int **vaudio_snd_prepare** (snd_pcm_substream∗ *substream*);

1.2.3.12.2 DESCRIPTION

This primitive updates rate, channel number and format and set the current command to *NK_VAUDIO_COMMA↩ ND_SET_RATE*. Then a cross interrupt is issued to the backend driver and handler through **vaudio_intr_ctrl**.

1.2.3.12.3 PARAMETERS

The single argument is a valid pointer to a *struct snd_pcm_substream* object (see **sound/pcm.h** file for further details).

1.2.3.12.4 RETURN VALUES

The returned code is got after invoking the frontend driver as explained below. In case of success, *NK_VAUDIO↩ _STATUS_OK* is returned, or *NK_VAUDIO_STATUS_ERROR* in case of error.

1.2.3.13 STARTING/STOPPING A PLAYBACK OR CAPTURE AUDIO STREAM

1.2.3.13.1 SYNOPSIS

Starting or stopping a playback or a capture audio stream is done by the **vaudio_snd_trigger** primitive.

#include <nk/nkern.h>
#include <vaudio.h>

int **vaudio_snd_trigger** (snd_pcm_substream∗ *substream*, int *cmd*);

1.2.3.13.2 DESCRIPTION

This primitive is able to start (*NK_VAUDIO_COMMAND_START*) a playback stream audio if the current stream identifier is set to *SNDRV_PCM_STREAM_PLAYBACK* or a capture audio stream otherwise. It is also possible to stop a previously playback or capture mode using the *SNDRV_PCM_TRIGGER_STOP*. In all the cases, the backend driver is invoked through a cross interrupt and handled through **vaudio_intr_ctrl**.

1.2.3.13.3 PARAMETERS

The first argument is a valid pointer to a *struct snd_pcm_substream* object. The second *cmd* argument holds either *SNDRV_PCM_TRIGGER_START* or *SNDRV_PCM_TRIGGER_STOP* valid commands.

1.2.3.13.4 RETURN VALUES

In case of success, 0 is returned, otherwise *EINVAL* is returned if the required command (that is, the second argument) is incorrect.

1.2.3.14 GETTING THE PROCESSED FRAME NUMBER

1.2.3.14.1 SYNOPSIS

The current frame number already processed is returned by the **vaudio_snd_pointer** primitive.

#include <nk/nkern.h>
#include <vaudio.h>

snd_pcm_uframes_t **vaudio_snd_pointer** (snd_pcm_substream∗ *substream*);

1.2.3.14.2 DESCRIPTION

This primitive returned the number of frames already processed in both playback and capture mode.

1.2.3.14.3 PARAMETERS

The single argument is a valid pointer to a *struct snd_pcm_substream* object (see **sound/pcm.h** file for further details).

1.2.3.14.4 RETURN VALUES

A *snd_pcm_uframes_t* number of frames is returned by this primitive. This type is an *unsigned long* value.

1.2.3.15 MAPPING THE DMA BUFFER

1.2.3.15.1 SYNOPSIS

The kernel substream dma buffer is remapped to a user address space.

#include <nk/nkern.h>
#include <vaudio.h>

int **vaudio_snd_mmap** (snd_pcm_substream∗ *substream*, struct vm_area_struct∗ *vma*);

1.2.3.15.2 DESCRIPTION

This primitive is based on the kernel Linux **remap_pfn_range**.

1.2.3.15.3 PARAMETERS

The first argument is a valid pointer to a *struct snd_pcm_substream* object (see **sound/pcm.h** file for further details) and the second argument is a valid pointer to a *struct vm_area_struct*.

1.2.3.15.4 RETURN VALUES

The result of **remap_pfn_range** is returned by this primitive.

1.2.3.16 SEE ALSO

vaudio_be

nk_id_get

nk_pdev_alloc

nk_ptov

nk_vtop

vlink_lookup

nk_xirq_attach

nk_xirq_detach

nk_xirq_trigger

1.2.4 DT(4D)

1.2.4.1 Cross References

Related Documents
Manual Page

1.2.4.2 NAME

dt — vlx device trees

1.2.4.3 SYNOPSIS

The vlx device tree driver (vlx-dt) permits to have a file system reprentation of DTBs (Device Tree Blobs) exposed by the hypervisor through DTB properties, in the same way as `/proc/device-tree`.

1.2.4.4 FEATURES

The vlx-dt driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> VLX device trees

It can be compiled as loadable module or as embedded driver.

1.2.4.5 DESCRIPTION

The provided user space interface will by a sysfs, located at the following path: /sys/nk/device-trees

The table below describes how device tree folder is named, depending on its related property.

DTB property name	device tree folder name
nk.vlm-dtb	vlm
nk.vm.<vmid>.vplatform-dtb	vplatform@<vmid>

For instance, assuming that a configuration is composed of 3 VMs, VM2, VM3 and VM4, and that the current VM has required permissions to access their related DTB properties and the vlm-dtb property, the tree folder would be like this:

```
/sys
└ nk
  └ device-trees          main vlx device trees folder
    ├ vlm                 device tree folder related to vproperty nk.vlm-dtb
    ├ vplatform@2         device tree folder related to vproperty nk.vm.2.vplatform-dtb
    ├ vplatform@3         device tree folder related to vproperty nk.vm.2.vplatform-dtb
    └ vplatform@4         device tree folder related to vproperty nk.vm.2.vplatform-dtb
```

Figure 1.1 Tree folder of device tree user interfaces

1.2.5 VM-MEMORY-HOTPLUG(4D)

1.2.5.1 NAME

VM Memory Hotplug.

1.2.5.2 SYNOPSIS

#include <vlx-memory-hotplug.h>

The `VM Memory Hotplug Driver` dynamically adds the memory blocks specified as the VM Hotplug memory in the Hypervisor configuration.

Thus, it allows to improve the kernel boot time KPIs by deferring the initialization of the Hotplug memory block's kernel memory management infrastructure such as physical page descriptors and kernel memory mappings.

1.2.5.3 FEATURES

The `VM Memory Hotplug Driver` should be enabled in the Linux configuration:

Device Drivers -> VLX virtual device support -> enable VM memory hotplug

It can be compiled as a loadable module or as a built-in driver.

Hotplug memory of different VMs is described through device-tree nodes present in the Guest OS device trees. These nodes must be compatible with *vl,vm-memory-hotplug*.

Here an example of such a node:

```
vm-memory-hotplug {
  compatible = "vl,vm-memory-hotplug";
  vl,vmid = <3>;
  reg = <...>;
  vl,prio = <10>;
};
```

All properties present in this node are required. Please find below their description:

- *vl,vmid:* an unsigned int. This is the vmid of the VM owning memory region described in the reg property. If it is different of the vmid of the current VM, it means that the described memory is imported.
- *reg:* standard reg field, listing memory regions that will be hotplugged by the `VM Memory Hotplug Driver`
- *vl,prio:* an unsigned int. It defines the priority of the node. The higher it is, the higher the priority of the node is. `VM Memory Hotplug driver` hotplugs memory described in these nodes by decreasing order of priority.

Note

The *reg* property is updated by the hypervisor. To do so, the hypervisor needs hotpluggable memory regions to be defined in the Hypervisor DT and/or virtual platform DT (please have a look the hypervisor reference manual for further details).

1.2.5.4 USER SPACE DESCRIPTION

On user space side, the `VM Memory Hotplug driver` offers a sysfs interface, permitting to trigger memory probing:

```
/sys
└ nk
  └ memory
    └ hotplug       VM memory hotplug directory
      ├ probe_all     Write Only: probe memory regions described in the reg property of all VM memory hotplug nodes
      ├ probe_vm      Write Only: when a vmid VMID is written in probe_vm, memory regions of the VM memory hotplug node of VM VMID are probed.
      └ status        Read Only: provides status of the memory hotplug (for debug purpose).
```

Figure 1.2 VM memory hotplug user space interface

1.2.5.5 KERNEL SPACE DESCRIPTION

On kernel side, it offers the following interface:

#include <vlx-memory-hotplug.h>

int **vlx_memory_hotplug_verify**(NkOsId *exporter_vmid*);

This function checks that /waits for all memory blocks reserved for all pages potentially imported from VM with vmid *exported_vmid* are online.

It returns 0 on success and a negative error code otherwise.

Please notice that the function can be called even if the `VM Memory Hotplug Driver` is not activated. In such a case, the function will return instantly.

1.2.5.6 USAGE

1.2.5.6.1 TRIGGER MEMORY HOTPLUG

The `VM Memory Hotplug Driver` relies on the Linux Kernel Memory Hotplug Framework. With this framework, memory hotplug is performed in two steps:

1. memory probing: memory is added to the system
2. memory onlining: memory is released, the system can use it for memory allocation

The `VM Memory Hotplug Driver` probe memory when its sysfs interfaces *probe_all* or *prob_vm* are used. It also online memory, but this operation has to be triggered externally.

Memory onlining can be triggered automatically by the Linux Kernel Memory Hotplug Framework after memory probing. It is possible to activate this behavior in difference ways:

- having *CONFIG_MEMORY_HOTPLUG_DEFAULT_ONLINE* set in the Linux kernel configuration
- writing *online* in the sys interface */sys/devices/system/memory/auto_online_blocks*

For instance, a simple way to trigger hotplug of the memory described in the `VM Memory Hotplug Driver` configuration is to have a service executing the following script at the boot of the system:

```
echo online > /sys/devices/system/memory/auto_online_blocks
echo 1 > /sys/nk/memory/hot-plug/probe_all
```

One should chose with caution the moment when such a service will be executed:

- on one hand, the later is the better, avoiding to disturb boot of other services
- on the other hand, ideally, memory hotplug should be finished before any driver using shared memory and having an activity visible by the final user start to work (typically graphical services), otherwise boot of these services will be delayed

Note

For further details about the Linux Kernel Memory Hotplug Framework, please have a look to the official Linux kernel documentation.

1.2.6 VBD2(4D)

1.2.6.1 NAME

vbd2 — Virtual Block Device v2 driver interface

1.2.6.2 SYNOPSIS

The virtual block device is an abstraction which enables a VM to access block devices such Hard Disks or Compact Disks managed by another VM. Either whole disks or individual partitions can be exported. Exported individual partitions can appear as whole disks in frontend, which allows to store full VM configurations inside a single backend disk partition.

This man page describes both the frontend and the backend virtual block device drivers (vbd2). The vbd2 backend driver receives read and write requests from the frontend vbd2 driver and forwards them to the underlying native block device driver via standard Linux kernel block device API. The vbd2 backend driver asynchronously informs its frontend counterpart when a read/write operation is completed. The frontend vbd2 driver exports the standard block devices to the local kernel.

1.2.6.3 FEATURES

The vbd2 frontend driver should be enabled in the Linux configuration file:

```
Device Drivers  ->  VLX virtual device support  ->  Virtual block device v.↩
2 frontend interface
```

The vbd2 backend driver should be enabled in the Linux configuration file:

```
Device Drivers  ->  VLX virtual device support  ->  Virtual block device v.↩
2 backend interface
```

Both drivers can be compiled as loadable modules or as embedded drivers.

Virtual block devices must be declared in the platform device tree. The example below declares vbd2 devices using the virtual link framework (see the NK_VLINK_LOOKUP(3D) manual page for details).

```
&vm2_vdevs {
    vbd_be: vbd@be {                     // vbd2 backend for VM3
        compatible = "vbd2";             // uses vbd2 protocol
        #clone      = <1>;               // 1 server & client end point
        info        = "16,,db,be",       // vbd2 backend driver parameters
            " bootargs:",                // prefix
            " vbd2_dma=0",
            " vbd2=(3,179,1:/dev/block/bootdevice/by-name/esystem,rw,nz)",
            " vbd2=(3,179,2:254,2,rw,nz)";
            " vbd2=(3,179,3:254,3,rw,nz)";
    };
};
&vm3_vdevs {
    vbd@fe {                             // vbd2 frontend
        peer-phandle = <&vbd_be>;        // peer vLINK
        info     =                       // vbd2 frontend options
                " bootargs:",            // prefix
                " vbd2-wait=(179,1)",
                " vbd2-wait=(179,2)",
                " vbd2-wait=(179,3)";
    };
};
```

There is only one virtual link in this example. The backend (server) side is managed by the backend vbd2 driver running in VM2. The frontend (client) side is managed by the frontend driver running in VM3.

The *info* property can be used on the backend side to change communications defaults and to define which devices should be exported.

The syntax for communications is:
`[msg_count][,[segs_per_req_max][,[db|odb[:data_buffers]|[pg[:off]|[:cpu|dma|read|write|nopar`

Parameter	Meaning
msg_count	Maximum number of parallel requests on the vlink. A request is a contiguous run of sectors. The default is 64.
segs_per_req_max	Maximum segments per request. A segment is a memory page or a fragment of it. The default is 128.
`db`	Enable *simple* double-buffering of disk buffers through PMEM.
`odb`	Enabled *optimized* double-buffering, with *data_buffers* value.
`pg`	Page granting control in zero-copy mode.
`be`	Identifies backend side of vlink in case of loopback configuration.

VBD2 backend uses various methods to access memory pages where frontend expects I/O data. Ideally, it can give the native disk driver direct access, achieving zero-copy operations. This requires that the hypervisor grants the backend VM access to frontend memory pages, and that the frontend memory pages are known to the backend VM (that is, have `struct page` descriptors), though are not necessarily mapped into kernel virtual address space.

If this cannot be achieved, double buffering must be used. It is performed through buffers located in PMEM memory area. The frontend is in charge of bouncing. This double buffering must be enabled explicitly, at the time when vlink resources are allocated, at it impacts the size of PMEM.

In the simple variant, the PMEM stores *msg_count* buffers, each having *segs_per_req_max* pages. With the default configuration values, this gives $64 * 128 * 4$ KiB or 32 MiB. Such large buffers are seldom used entirely, wasting memory.

When optimized double-buffering is enabled, bouncing buffers for requests are allocated using several page-sized buffers in PMEM, instead of just one large buffer of maximum size. The number of page-sized buffers depends on the size of each request.

To activate optimized double-buffering, `odb` should be passed in place of `db`. By default, vbd2 backend will use 4 times less buffer memory than with ordinary double-buffering (8 MiB). One can optionally force the number of (page-sized) data buffers to be used, for example to be even smaller. This number will be adjusted up if necessary, so that at least 2 maximum size requests can be performed in parallel. Therefore the memory usage formula becomes: maximum of *msg_count* times *data_buffers* and of `2` times `segs_per_req_max` pages.

In the previous device tree example, the `vbd_be` device is a backend using simple double-buffering and 16 as the maximum number of parallel requests on the vlink. The pool of requests is shared by all the disks exported towards VM3.

Zero-copy mode is turned on implicitly when the double-buffering keywords `db` and `odb` are omitted from the *info* parameters string. The frontend driver automatically activates the hypervisor page granting mechanism (if it is enabled in hypervisor) in this case, using default parameters. The `db` and `odb` keyword can also be replaced with the page granting `pg` keyword in order to tune the page granting behavior, as follows:

```
pg[:off]|[:cpu|dma|read|write|nopanic[+...]]
```

Where:

- `pg:off`: deactivates page granting even if it is supported by hypervisor.
- `pg:cpu|dma|read|write|nopanic[+...]`: activates page granting with specific attributes:
  - `cpu`: allow CPU access
  - `dma`: allow DMA access
  - `read`: allow read access (even on vbd2 read)
  - `write`: allow write access (even on vbd2 write)

  - `nopanic`: no panic on denying failure of previously granted pages
  - multiple attributes can be combined using +

- `pg` - activate the page granting with default attributes:
  - `dma+read` for vbd2 write
  - `dma+write` for vbd2 read

The following example allows the backend side to perform both CPU and DMA accesses to remote buffers, which are always readable (even on vbd2 read).

```
pg:cpu+dma+read
```

The `bootargs:` prefix in the backend *info* property starts the blank-delimited list of virtual disks to be exported to the frontend drivers, and other options. The syntax to export a single block device is:
`vbd2=(`*owner*`,`*vmajor*`,`*vminor*`{:|/}{`*lmajor*`,`*lminor*`|`*path*`},{ro,rw}[,[nw|wa|nz]])`

Parameter	Meaning
owner	Virtual disk VM owner, should be the same as the number of the VM on the frontend side of the vlink
vmajor	Virtual disk major number in frontend VM
vminor	Virtual disk minor number in frontend VM
lmajor	Local disk major in backend VM
lminor	Local disk minor in backend VM
path	Pathname of the local disk device in backend VM
ro	Read-only disk
rw	Read-write disk
nw	Do not wait for this virtual disk to exist before declaring it as available to frontend
wa	Wait for this virtual disk
nz	Wait for this virtual disk to be non-zero sized

The `vbd2_dma=`*value* parameter can be added to choose DMA mode I/O (value 1) or copy mode I/O (value 0), the default being DMA mode. DMA transfers are performed if possible, otherwise data are copied and this option forces them all to copy mode. They are copied either directly from frontend VM pages or from bouncing buffers in PMEM. This option is global to all vlinks.

The *info* property can also be used on the frontend side to be able to wait for the start of vbd2 devices on the backend side, blocking the initialization of the frontend driver. The syntax is:
`[[bootargs: ]{[ vbd2-wait=(`*major*`,`*minor*`)]}]`

In the previous device tree example, the frontend will wait until disks with major 179 and minors 1, 2 and 3 are ready on the backend side.

1.2.6.4 IMPLEMENTATION

The virtual block device backend and frontend drivers use the Virtual Message Queue (VMQ(4D)) driver for remote operations between VMs.

The frontend driver is acting as a transmitter and thus *transmit* configuration parameters are provided by this driver. At initialization time, `vmq_links_init_ex()` is invoked to setup the vmq callback routines and receive configuration.

The backend driver is acting as a receiver and thus *receiver* configuration parameters are provided by this driver. At initialization time, `vmq_links_init_ex()` is invoked to setup the vmq callback routines and transmit configuration.

There are five main remote disk operations supported by a backend virtual block device for frontend drivers:

Name	Description
`VBD2_OP_PROBE`	Probe a virtual disk.
`VBD2_OP_READ`	Read one or multiple sectors from a virtual disk.
`VBD2_OP_WRITE`	Write one or multiple sectors to a virtual disk.
`VBD2_OP_CHANGES`	Signal that a change has occurred in the virtual disks configuration
`VBD2_OP_FLUSH_DISK_CACHE`	Flush the virtual disk.

A `vbd2_req_header_t` object is shared between frontend drivers to send requests to a backend driver. The layout of this object is shown below:

```
typedef nku16_f vbd2_devid_t;
typedef nku16_f vbd2_genid_t;
typedef nku64_f vbd2_sector_t;
typedef nku64_f vbd2_cookie_t;
typedef nku8_f  vbd2_op_t;
typedef nku8_f  vbd2_status_t;
typedef nku8_f  vbd2_count_t;
    /*
     * Remote Disk request header.
     * Buffers follow just behind this header.
     */
typedef struct vbd2_req_header_t {
    vbd2_cookie_t cookie;   /* (64b) to put in the response descriptor */
    vbd2_sector_t sector;   /* (64b) sector */
    vbd2_devid_t  devid;    /* (16b) device ID */
    vbd2_op_t     op;       /*  (8b) operation */
    vbd2_count_t  count;    /*  (8b) number of buffers which follows */
    vbd2_genid_t  genid;    /* (16b) generation of devid */
    nku16_f       reserved; /* (16b) reserved */
} vbd2_req_header_t;
```

A `vbd2_resp_t` is always returned by the backend driver for all operations: read, write, and probe. In addition, for this later operation a `vbd2_probe_t` is returned by the backend driver. The layout of these objects is defined below:

```
#define VBD2_STATUS_OK      0
#define VBD2_STATUS_EOPNOTSUPP  0xfe
#define VBD2_STATUS_ERROR   0xff
    /*
     * Response descriptor
     * status is the "count" field.
     */
typedef struct vbd2_req_header_t vbd2_resp_t;
    /*
     * Probing record
     */
typedef nku16_f vbd2_info_t;
typedef nku16_f vbd2_xinfo_t;

typedef struct vbd2_probe_t {
    vbd2_devid_t  devid;    /* device ID (16 bits) */
    vbd2_info_t   info;     /* device type & flags (16 bits) */
    vbd2_genid_t  genid;    /* generation of devid (16 bits) */
    vbd2_xinfo_t  xinfo;    /* more flags (16 bits) */
    vbd2_sector_t sectors;  /* size in sectors (64 bits) */
} vbd2_probe_t;
```

1.2.6.4.1 Probing virtual disks

The frontend driver invokes this service at initialization time to check if disks are running and to get their size, expressed in number of sectors of 512 bytes each. Virtual disks are probed by filling a `vbd2_req_header_t` object and getting a response through `vbd2_probe_t` objects, according to the number of available virtual disks. A message is allocated using `vmq_msg_allocate()`, the header being a `vbd2_probe_link_t`:

```
typedef struct vbd2_probe_link_t {
    vbd2_req_header_t   common;
    vbd2_probe_t        probe[1];
} vbd2_probe_link_t;

#define VBD_LINK_MAX_DEVIDS_PER_PROBE(vbd) \
    (((vbd)->msg_max - sizeof (vbd2_req_header_t)) / sizeof (vbd2_probe_t))
```

The `vbd2_req_header_t` object is filled with the following values:

Field	Value
op	`VBD2_OP_PROBE`
count	`VBD_LINK_MAX_DEVIDS_PER_PROBE(`*vbd*`)`

All other fields are set to zero.

Then, this message is sent to the backend using `vmq_msg_send()`. When the command has been processed by the backend, the *count* is updated and contains the result of the probe, which can be `VBD2_STATUS_ERROR` if the request failed, or the number of virtual disks detected. In case of success, the *count* disks are initialized, and finally, the message is released using `vmq_return_msg_free()`.

1.2.6.4.2 Read/write operations to/from a virtual disk

The frontend driver invokes standard I/O operations after probing a virtual disk to perform I/O. Virtual disk are read/written by filling a `vbd2_req_header_t` object and getting a response after operation has completed. A message is allocated using `vmq_msg_allocate()`. The `vbd2_req_header_t` object of this message is filled with the following values:

Field	Value
op	`VBD2_OP_WRITE` or `VBD2_OP_READ`, according to the required I/O operation
cookie	Internal data belonging to the frontend driver and allowing to identify the I/O request in case of asynchronous behavior
sector	Starting sector on the virtual disk
devid	Device identifier previously returned by a probe operation
genid	Generation identifier previously returned by a probe operation
count	Number of segments in request

All other fields are set to zero.

This message is sent to the backend asynchronously using `vmq_msg_send_async()` and is flushed using `vmq_msg_send_flush()` when Linux has finished submitting the batch of requests. When the command has been processed by the backend, a return message is sent to the frontend.

1.2.6.4.3 Virtual disk changes operation

This operation is an asynchronous event from the backend driver to the frontend driver in order to signal a change in the virtual disks configuration. The frontend will probe the virtual disks again.

1.2.6.4.4 Virtual disk flush cache operation

This operation forces a disk cache flush of a virtual disk before subsequent requests are carried.

1.2.6.5 SEE ALSO

VMQ(4D)

VLINK_LIB(4D)

1.2.7 VBPIPE(4D)

1.2.7.1 NAME

vbpipe — Virtual Bidirectional Pipe

1.2.7.2 DESCRIPTION

The Virtual Bidirectional Pipe feature provides a bidirectional communication link between Linux user space applications running in two different Virtual Machines. The semantics provided is identical to that of a Unix pipe. However, since it is bidirectional, it is closer to BSD pipes.

The service is provided by a Linux kernel module. Each vbpipe driver acts simultaneously as a front-end and as a back-end driver. It relies on the Hypervisor vlink service.

1.2.7.3 CONFIGURATION

1.2.7.3.1 Linux Kernel Configuration

The virtual bidirectional pipe driver should be enabled in the Linux kernel configuration file:

Device Drivers -> VLX virtual device support -> Virtual bidirectional pipe for inter OS communication

It can be compiled as a loadable module or as an embedded driver.

1.2.7.3.2 Device Tree Configuration

Virtual bidirectional pipe devices must be declared in the platform device tree. The example below declares vbpipe devices using the virtual link framework (see the nk_vlink_lookup manual page for more details).

```
&vm2_vdevs {
    vblobmgr_be: vblobmgr@be {                      // vBlobManager back-end for VM3
        compatible = "vbpipe";                      // uses generic vBPIPE protocol
        info = ";;;wakeup=250;name=vblobManager";
    };
};

&vm3_vdevs {
    vblobmgr@fe {                                   // vBlobManager front-end
        compatible = "vbpipe";                      // uses generic vBPIPE protocol
        peer-phandle = <&vblobmgr_be>;              // peer vLINK
        info = ";;;wakeup=250;name=vblobManager";
    };
};
```

The syntax and semantics of the "info" field is as follows:

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

where info is a string with the following fields:

Parameters

a	wait-only-on-empty policy : when this flag is configured, the read system call only blocks when the receive buffer is empty, as a consequence, the read system call might return less data than what is specified by the third parameter.
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	This option allow to configure the minor number which will be used by vbpipe device driver.
msecs	number of milliseconds during which a wake-lock should be held following reception of data
dev-name	a customized device name (default is /dev/vbpipeX where X is an automatically assigned number)

1.2.7.4 USER SPACE DESCRIPTION

vbpipes appear as character devices in the Linux filesystem. The nodes are created by the vbpipe driver. A Linux user space application can then use regular file system calls such as open(2), read(2), Write(2) and close(2) to receive or send data through the pipe to another user space application running in a possibly different Virtual Machine.

1.2.7.5 /proc/nk Entries

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:

Parameters

Mi	minor device number 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.

1.2.7.6 KERNEL DESCRIPTION

1.2.7.6.1 SYNOPSIS

#include <nk/nkern.h>

The virtual bidirectional pipe driver runs as a Linux kernel driver. This driver is given as an example, and can be adapted to run on top of realtime OS-es. It is based on two unidirectional communication lines in opposite directions. Because of that, the vbpipe driver acts as both frontend and backend driver.

1.2.7.6.2 OPENING A BIDIRECTIONAL PIPE

1.2.7.6.2.1 SYNOPSIS

```
unsigned int ex_open (struct inode* inode, struct file* file);
```

Before using a virtual bidirectional pipe, it must be opened by invoking the **ex_open** primitive. This operation is executed by the Linux kernel when an application performs **open** system call.

1.2.7.6.2.2 DESCRIPTION

The **ex_open** primitive is responsible for opening a virtual bidirectional pipe. Upon the first open operation, the **ex_xirq_hdl** cross interrupt handler is attached for both client and server sides. At the same time a synchronization (vlink handshake) with the peer driver is done through the Linux kernel primitive **wait_event_freezable**. Multiple open operations are allowed on a same virtual pipe (that is, same couple of *inode*, *file* parameters). In that case, the internal counter of *ExDev* object is incremented, no additional synchronization with peer driver is performed. The peer driver synchronization is implemented in the internal **ex_link_ready**) function. It consists in starting up the virtual link handshake protocol to establish communication between the backend and its peer counterpart (that is, the frontend driver) (see nk_vlink_lookup manual page for further details).

Parameters

inode	is a valid pointer to a Unix inode (system data part associated with a file) defined in **linux/fs.h**.
file	is a pointer to a file data structure also defined in **linux/fs.h** file containing all data related to a file descriptor.

Return values

0	In case of success, this primitive returns 0, otherwise one of the following error codes is returned:
ENXIO	is returned in case of incorrect minor number extracted from the caller's *inode* parameter, or when the underlying initialization (**ex_dev_init()**) has failed;
ENOMEM	is returned if the **ex_xirq_hdl()** cross interrupt handler cannot be connected. This error code is returned only upon the first open operating on the virtual bidirectional pipe;
EINTR	is returned if frontend/backend driver synchronization is interrupted by a signal.

1.2.7.6.3 RELEASING OR CLOSING A BIDIRECTIONAL PIPE

1.2.7.6.3.1 SYNOPSIS

```
unsigned int ex_release (struct inode* inode, struct file* file);
```

A virtual pipe is shut down or closed when **ex_release** is invoked. This operation is executed by the Linux kernel when an application performs **close** or **exit** system calls.

1.2.7.6.3.2 DESCRIPTION

The **ex_release** implements a close or a release operation required by Linux semantic for character devices in character mode. In case of errors for read/write operations (**ex_read**, **ex_write**), an application can execute **close** system call to to shut the link (virtual bidirectional pipe) down and to set the link state to *NK_DEV_VLINK_OFF*.

As **ex_open** can be invoked multiple times on a same virtual pipe, the close operation is effectively done when the counter described in *ExDev* object reaches zero.

When the counter reaches zero, the **ex_xirq_hdl** is detached through **ex_dev_cleanup** (see nk_xirq_detach manual page for further details) and the status of the current mode: client and server states are set to *NK_DEV_VL*↩ *INK_OFF*. Finally a *NK_XIRQ_SYSCONF* is triggered through the **nk_xirq_trigger** primitive in order to close both server server and client sides of the virtual bidirectional pipe (see nk_xirq_trigger manual page for further details). The **ex_sysconf_trigger** subroutine is responsible for triggering the *NK_XIRQ_SYSCONF* cross interrupt, to inform the peer driver that the state of the vlink is changed. The **ex_handshake** function is responsible for processing this cross interrupt.

Parameters

inode	is a valid pointer to a Unix inode (system data part associated with a file) defined in **linux/fs.h**.
file	is a pointer to a file data structure also defined in **linux/fs.h** file containing all data related to a file descriptor.

Return values

0	In case of success, this primitive returns 0 otherwise,
EINTR	is returned if this operation is aborted because of signal while waiting on a mutes protecting the corresponding *ExDev* data structure.

1.2.7.6.4 READING FROM A BIDIRECTIONAL PIPE

1.2.7.6.4.1 SYNOPSIS

```
unsigned int ex_read (struct file* file, char __user* buf, size_t count, loff_t* ppos);
```

Characters are read from a virtual bidirectional pipe using **ex_read** primitive. This operation is executed by the Linux kernel when an application performs **read** system call.

1.2.7.6.4.2 DESCRIPTION

This primitive implements a read operation required by Unix semantic for character devices. The **ex_read** attempts to read all *count* bytes from the writer side (client). If the underlying circular buffer is empty, this routine waits for characters if **ex_open** has been called without the bit *O_NONBLOCK* set in **file->f_flags** (see the Linux/Unix manual page of POSIX **open()** for further details). If the circular buffer becomes non full after **ex_read**, a cross interrupt is sent to the client side in order to continue write operations.

If partial read semantic is allowed for a particular vbpipe device ("|a" string is put after link number in the command line for this vbpipe), then **ex_read** reads only currently available characters. It waits for characters only if there is no available characters at all (the circular buffer is empty).

In case of non blocking I/O (*O_NONBLOCK* bit is set), the caller is never blocked. The **ex_read** returns the number of characters read from the circular buffer. If the circular buffer was empty **ex_read** returns *EAGAIN*.

Parameters

file	is a valid pointer to a file data structure defined in **linux/fs.h** file containing all data related to a file descriptor.
buf	is a valid pointer in user space (the *__user* macro is defined in file **linux/compiler.h** indicating to the C GNU compiler that the pointer is belonging to the user virtual space) for storing characters.
count	parameter is the number of required bytes to read and *ppos* is not used in this driver.

Returns

In case of success, the number of read bytes is returned. This number can be equal or less that the required *count* given by the caller. In case of error, the following error codes are returned:

Return values

EAGAIN	is returned if no characters have been read while the current link status is still alive (that is, equal to *NK_DEV_VLINK_ON*) and the read is in non blocking mode (bit *O_NONBLOCK* set to one);
EFAULT	is returned if the *buf* is an invalid user address;
EINTR	is returned when current read operation is aborted because of signal.

1.2.7.6.5 WRITING TO A BIDIRECTIONAL PIPE

1.2.7.6.5.1 SYNOPSIS

```
unsigned int ex_write(struct file* file, char __user* buf, size_t count, loff_t* ppos)
```

Characters are written to a virtual pipe using **ex_write** primitive. This operation is executed by the Linux kernel when an application performs **write** system call.

1.2.7.6.5.2 DESCRIPTION

This primitive implements a write operation required by Unix semantic for character devices. **ex_write** transfers all *count* bytes to the reader side (server). If the underlying circular buffer is full, this routine waits until there is some room to store the required characters. If the circular buffer becomes non empty a cross interrupt is sent to the server side in order to continue read operations.

In the case of non blocking I/O (*O_NONBLOCK* bit is set), the caller is never blocked. The "small" writes (with the size less or equal to the size of circular buffer) are never partial: if there is enough room in the circular buffer all bytes would be written, otherwise no bytes would be written at all and **ex_write** returns *EAGAIN*. The "big" writes are always partial. The **ex_write** returns how many bytes were successfully written. If the circular buffer is full **ex_write** returns *EAGAIN*.

Parameters

file	is a valid pointer to a file data structure defined in **linux/fs.h** file containing all data related to a file descriptor.
buf	is a valid pointer in user space (the *__user* macro is defined in file **linux/compiler.h** indicating to the C GNU compiler that the pointer is belonging to the user virtual space) for storing characters.
count	is the number of required bytes to write.
ppos	is not used in this driver.

Returns

In case of success, the returned value is equal to the required *count* given by the caller. In case of error, the following error codes are returned:

Return values

EPIPE	if the current state of the server side is no longer equal to *NK_DEV_VLINK_ON*;

Return values

EAGAIN	is returned if the device has been opened with *O_NONBLOCK* bit set and if there is no enough room in the underlying circular buffer for "small" writes (size less or equal to the circular buffer size) or if the circular buffer is full for "big" writes;
EFAULT	is returned if the *buf* is an invalid user address;
EINTR	is returned when current write operation is aborted because of signal.

1.2.7.6.6 POLLING FROM A BIDIRECTIONAL PIPE

1.2.7.6.6.1 SYNOPSIS

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

The **ex_poll** checks if any characters can be read or if there is enough room to write some characters on a virtual pipe. This operation is executed by the Linux kernel when an application performs system calls like **select**.

1.2.7.6.6.2 DESCRIPTION

This primitive implements a poll operation required by Unix semantic for character devices. It is responsible to check if there are any characters to read from the virtual pipe, or if there is any room to write characters to the virtual pipe. Consequently, this primitive can be call from both side: client or server and the result is returned accordingly.

Parameters

file	is a valid pointer to a file data structure defined in **linux/fs.h** file containing all related to a file descriptor.
wait	is a valid pointer to a *poll_table* object defined in **linux/poll.h** file. This data structure is used by the kernel Linux primitive **poll_wait** also defined in the same file.

Return values

null	A null value is returned from the client side if there is no characters to read from the virtual pipe, or from the server side if there is no room to write at least one character to the virtual pipe.

Returns

A non zero value is returned on the client side if at least one character can be read from the virtual pipe. In that case, the error code holds two bits set *POLLIN* and *POLLRDNORM*. These bits are defined in **linux/asm/poll.h** file.

A non zero value is returned on the server side if at least one character can be written to the virtual pipe. In that case, the error code holds two bits set *POLLOUT* and *POLLWRNORM*. These bits are defined in **linux/asm/poll.h** file.

1.2.7.7 IMPLEMENTATION DESCRIPTION

This driver uses a couple of unidirectional links to implement a bidirectional virtual pipe. Each link is a simple circular buffer of characters using free running indexes for the producer which is writing characters to the client circular buffer, and for the consumer which is reading characters from the server circular buffer. Cross interrupts are sent to alert each peer driver (producer and consumer) when that circular buffer became either non empty or non full.

By convention a driver connected to a client side of a communication link (it is also called frontend driver) puts/writes characters into this circular buffer, and a driver connected to a server side of a communication link (it is also called backend driver) gets/reads characters from this circular buffer.

A circular buffer is also called ring buffer and located in the shared persistent memory (see nk_pmem_alloc and nk_mem_map manual pages for further details) and thus visible to both sides of the link. The layout of this buffer is a *ExRing* object and has the following layout:

```
@#define DEF_RING_SIZE   0x1000
@#define MIN_RING_SIZE      16

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 **s_idx** variable is a "free running" server index. It is incremented by the backend driver (server) each time when it reads characters from the circular buffer. It is never decremented. To use it as a buffer index, it should be get modulo buffer size. The **c_idx** variable is a "free running" client index. It is incremented by the frontend driver (client) each time when it writes characters to the circular buffer.

When several VMs are run on top of the Hypervisor, there is no synchronization between them. In addition, the Hypervisor can switch to another VM at any moment. The fields **s_idx** and **c_idx** have a volatile attribute since they can be changed by a peer driver at any moment, so the C compiler is not allowed to optimize these fields accesses.

The virtual Pipe driver uses cross interrupts mechanism provided by the device driver framework to wake its peer driver up when some work must be done either to read some characters from the link by the consumer (because the producer has written some characters to it), or to write some characters to the link by the producer (because the consumer has read some characters from it).

An optimization is provided for the virtual bidirectional pipes in order to minimize cross interrupt traffic. The fields **s_wait** and **c_wait** are used respectively to record that a server is going to sleep (that is, to wait for characters to be read) and a client is going to sleep (that is, wait for a free space to write characters).

Macros are provided to manage the available room in a ring buffer from both consumer and producer sides and are described in the vpipe manual page at EXTENDED DESCRIPTION section.

For each vbpipe device the virtual pipe driver has a private (not visible by its peer driver) data structure describing this device. It is allocated and initialized when the device driver is loaded as a module or when the Linux kernel is booted if the driver is compiled as an embedded one. The layout of a *ExDev* whose members are described below:

```
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;

@#define EMPTY_WAIT_ONLY   0x1  /* The read wait only on empty buffer */
```

The virtual bidirectional pipe driver works with a couple of links. They are referred to as client link (that is, this driver is connected to its client side), and as server link (that is, this driver is connected to its sever side). Similarly, the corresponding circular buffers or ring buffers as client ring (that is, ring used by a client link) and as server ring (that is, ring used by a server link). This symmetrical communication link actually uses two asymmetrical virtual communication links (virtual links or vlink) working in opposite directions.

The cross interrupt identifiers on the server ring and on the client ring to attach their respective handler are respectively held in *s_s_xid* and *c_c_xid* fields in order to be detached (see nk_xirq_attach and nk_xirq_detach manual pages for further details).

The *enabled* field is set to 1 when the *ExDev* is correctly allocated and all fields have been successfully initialized. The fields *s_s_xirq* and *s_c_xirq* hold respectively the virtual interrupt number for a server cross interrupt accessing the server ring, and the virtual interrupt number for a client cross interrupt for the server ring (see nk_pxirq_alloc manual page for further details).

The fields *c_s_xirq* and *c_c_xirq* respectively holds the virtual interrupt number for a server cross interrupt accessing the client ring, and the virtual interrupt number of a client cross interrupt accessing a client ring.

The vbpipe device driver registers its devices as a regular character devices, so an user application can use standard system calls as open, close, read, write and select. Note that lseek system call is not allowed for an obvious reasons.

The vbpipe device driver exports the following basic operation to the generic character device framework available in the Linux kernel: **ex_open**, **ex_release** **ex_read**, **ex_write** and **ex_poll**. All of them (except **ex_poll**) use a mutual exclusion mechanism to ensure that only one thread is performing a service at time.

A single waiting queue is implemented for all blocking operations (*wait*). All virtual interrupt handlers for cross interrupts ((**ex_xirq_hdl** and *NK_XIRQ_SYSCONF* interrupts (**ex_sysconf_hdl**, see **nk_xirq_trigger(3D)** manual page for further details) always wake up all pending threads to execute handler processing. The sleeping primitive is implemented through the Linux kernel primitive **wait_event**, so awaken threads will recheck its sleeping conditions and perform appropriate actions. In addition, all sleeping conditions always check the peer driver status. If this status is not set to *NK_DEV_VLINK_ON* state, the current pending operation is aborted.

The *count* field is used to keep track of multiple **ex_open** operations on the same vpipe.

1.2.7.8 SEE ALSO

nk_pmem_alloc

nk_mem_map

nk_vtop

nk_ptov

nk_vlink_lookup

nk_xirq_trigger

nk_pxirq_alloc

nk_xirq_attach

nk_xirq_detach

1.2.8 VBUFQ(4D)

1.2.8.1 Cross References

Related Documents
Manual Page

1.2.8.2 NAME

vbufq — Virtual Buffer Queue Drivers

1.2.8.3 SYNOPSIS

The virtual buffer queue drivers (`vbufq`) permit to issue requests to the native buffer queue driver (`bufq`) remotely from another VM. They are composed of a backend driver and of a frontend driver. The backend driver has to be in a VM where an instance of the `bufq` native driver is present.

1.2.8.4 FEATURES

The `vbufq` drivers should be enabled in the Linux configuration:

Device Drivers -> VLX virtual device support -> VLX virtual buffer queue drivers

They can be compiled as loadable modules or as embedded drivers.

`vbufq` drivers must be declared in the platform device tree. The example below declares `vbufq` devices using the virtual link framework (see NK_VLINK_LOOKUP(3D) for more details).

```
&vm2_vdevs {
    vbufq_be: vbufq@be {                    // vbufq backend
        compatible = "bufq";                // uses generic bufq protocol
        info       = "be,32;my_dev_name";   // configuration
    };
};
&vm3_vdevs {
    vbufq@fe {                              // VM3 vbufq frontend
        peer-phandle = <&vbufq_be>;         // peer vLINK
        info         = "";                  // unused by the frontend
    };
};
```

There is only one link in this example. The backend (server) side is managed by the backend `vbufq` driver running in VM2. The frontend (client) side is managed by the `vbufq` frontend driver running in VM3. The *info* property must be used on the backend side to set configuration.

The syntax is: `be,`*`msg_count`*`,`*`device_name`*

- `be`: Identifies backend side of vlink in case of loopback configuration (a local `vbufq` frontend exists)
- *`msg_count`*: Maximum number of requests on the vlink
- *`device_name`*: Name of the character device that will be created in `/dev`, permitting to reach the frontend driver

It is possible to connect a frontend to multiple backends and multiple frontends to a backend.

The `vbufq` backend driver optionally supports running in Google's Generic Kernel Image mode, where it is loaded dynamically and only uses a Google-approved subset of exported Linux kernel symbols. For this, it must be compiled with the `CONFIG_VLX_VBUFQ_BE_VGKI` configuration option. It then uses the VGKI kernel-mode API instead of non-GKI calls, requires the presence of the VGKI(4D) driver to load and of the vgki-helper(8) process to perform requests. The `vbufq` frontend driver is GKI compatible by default.

1.2.8.4.1 MSG_COUNT TUNING

The *`msg_count`* parameter tunes the maximum number of requests that can be issued simultaneously by user applications on the frontend. Thus, it must be properly sized in order to:

- avoid slowing down the request throughput (requests will be queued if *`msg_count`* is reached)
- avoid to consume too much memory

A good rule of thumb is that *`msg_count`* value should be at least the maximum number of threads that will use the frontend interface at the same time. For instance, if there are 5 processes, each with 2 threads which use a `vbufq` frontend interface, *`msg_count`* could be 10.

The drivers will round up the *`msg_count`* value to highest power of 2. Considering the previous example, the final value of *`msg_count`* will be 16.

1.2.8.5 SEE ALSO

VMQ(4D) - a driver internally used by the `vbufq` drivers in order to provide remote communications between VMs

VGKI(4D) - a driver internally used by `vbufq` backend for Generic Kernel Image compatibility

1.2.9 VCLK(4D)

1.2.9.1 Cross References

Related Documents
Manual Page

1.2.9.2 NAME

vclk — Virtual Clock Drivers

1.2.9.3 SYNOPSIS

The virtual clock (vclk) back-end and front-end drivers, when paired together, allow Linux clock consumers defined in one VM to reference Linux clock providers defined in another VM.

1.2.9.4 FEATURES

Under Linux, sources of clock signal are typically represented by nodes in the device tree. Those nodes are called **clock providers**. Device nodes that depend on those sources are called **clock consumers**. For a general overview of those concepts, please refer to `Linux clock device tree binding.`

Virtual clock back-end driver's responsability is to export local clock providers to other VMs. It can be enabled in the Linux kernel configuration:

Device Drivers -> VLX virtual device support -> Virtual Clock back end driver

Virtual clock front-end driver's responsability is to allow local clock consumers to reference clock providers exported by other VMs. It can be enabled in the Linux kernel configuration:

Device Drivers -> VLX virtual device support -> Virtual Clock front end driver

Virtual clock back-ends and front-ends communicate using VRPC. One VRPC link must be declared in the platform device tree for each (back-end, front-end) pair that shares clocks. The example below declares one vrpc link (VM2, VM3) using the virtual link framework (see vrpc manual page for more details).

```
&vm2_vdevs {
   vclk_be: vclk@be {                // vclk back-end
        compatible = "vrpc";         // uses generic vRPC protocol
        server;                      //
        info       = "vclk_ctrl";    // driver name
   };
};

&vm3_vdevs {
    vclk@fe {                        // VM3 vclk front-end
        peer-phandle = <&vclk_be>;   // peer vLINK
        client;                      // front-end end point
        info         = "vclk_ctrl";  // driver name
    };
};
```

Below is a typical example of a clock consumer using signals of a clock provider:

```
// Native Linux DTS.

    // clock provider A
    my_clk_providerA: pllA {
        compatible = "vendor,some-clk-controllerA";
        #clock-cells = <2>;
    };

    // clock provider B
    my_clk_providerB: pllB {
        compatible = "vendor,some-clk-controllerB";
        #clock-cells = <1>;
    };

    // clock consumer
    my_clk_consumer: devx {
        compatible = "vendor,some-dev-controller";
        clocks = <&my_clk_providerA 433 26>, <&my_clk_providerB 132>;
        clock-names = "mainclk", "auxclk";
    };
```

Below example shows how above configuration can be modified to move clock consumer to another VM.

```
// VM2 Linux DTS configuring back-end.

    // clock provider A
    my_clk_providerA: pllA {
        compatible = "vendor,some-clk-controllerA";
        #clock-cells = <2>;
    };

    // clock provider B
    my_clk_providerB: pllB {
        compatible = "vendor,some-clk-controllerB";
        #clock-cells = <1>;
    };

    // clock consumer: disabled but kept for parsing.
    my_clk_consumer: devx {
        compatible = "disabled";
        clocks = <&my_clk_providerA 433 26>, <&my_clk_providerB 132>;
        clock-names = "mainclk", "auxclk";
    };

    /*
      Node describing clock signals that are exported to VM3.
      Device tree may contain multiple "vl,vclk-be" compatible nodes,
      one per peer VM.
    */
    vlx-clk-be@3 {

                compatible = "vl,vclk-be";

                // This node describes clocks exported to VM3.
                vl,vm-id = <3>;

                // We export two clk providers to VM3.
                vl,clock-providers = <&my_clk_providerA>, <&my_clk_providerB>;

                // Names identifying the providers. Those will be used by
                // vclk front-end in VM3.
                vl,clock-provider-names = "my_clk_providerA", "my_clk_providerB";

                // We only export the signals used by one or several clock
                // consumer.
                clk@0 {
                    vl,clock-ref = <&my_clk_consumer>;

                    // You could further limit the export to signals specified
                    // by names, adding this optional property.
                    // vl,clock-names = "mainclk";
```

```
            };

            // You could add other consumers here.
            // clk@1 {
            //    vl,clock-ref = <&my_other_clk_consumer>;
            // };

    };
```

Next step is to configure a virtual clock front-end in VM3 to access the exported clock signals, as seen in below example:

```
// VM3 Linux DTS configuring front-end.

    // clock provider A
    my_clk_providerA: pllA {
        compatible = "vl,vclk-fe";
        vl,clock-provider-name = "my_clk_providerA";
        #clock-cells = <2>;
    };

    // clock provider B
    my_clk_providerB: pllB {
        compatible = "vl,vclk-fe";
        vl,clock-provider-name = "my_clk_providerB";
        #clock-cells = <1>;
    };

    // clock consumer
    my_clk_consumer: devx {
        compatible = "vendor,some-dev-controller";
        clocks = <&my_clk_providerA 433 26>, <&my_clk_providerB 132>;
        clock-names = "mainclk", "auxclk";
    };
```

In above example, clock consumer is defined as usual, but clock providers have to rely on device tree binding "vl,vclk-fe". Each clock provider must include a property "vl,clock-provider-name" set to back-end provider name, as it was specified by "vl,clock-provider-names" in the back-end device tree.

1.2.9.5 NOTES

An empty property *vl,force-enable* can be added right next to the property *vl,clock-ref* to always prepare and enable the referenced clocks.

Furthermore, an empty property *vl,s2r-suspend* can be added to disable and unprepare the referenced clocks when the back-end VM suspends.

Symmetrically, an empty property *vl,s2r-resume* can be added to prepare and enable the referenced clocks when the back-end VM resumes.

An extra option in Linux kernel configuration also allows you to always enable *all* the clock sources identified by the virtual clock back-end driver at boot time:

Device Drivers -> VLX virtual device support -> Enable all exported clocks at the boot time

By default, the virtual clock driver is using the RPC channel *vclk_ctrl* to remotely execute all clock operations in a kernel thread of the back-end. There is an exception to this rule: *enable* and *disable* operations are not handled like that because they are supposed to be atomic. In its default implementation, virtual clock driver implements *enable* and *disable* operations as empty callbacks and performs the real *enable/disable* work at the end of the *prepare* and at the beginning of the *unprepare* operations. There might be corner cases where this default behavior causes issues, so the user can change it on a per-clock basis switching to a mode that uses a second RPC channel to really execute the *enable/disable* operations on the back-end in interrupt context. To select this mode, the user must define an extra VRPC channel for the targeted (back-end, front-end) pair named *vclk_fast_ctrl*, and add a property named *vl,use-fast-rpc* right next to the *vl,clock-ref* property referencing the targeted clocks.

Below is an example of such configuration:

```
&vm2_vdevs {

    // Default RPC channel back-end for virtual clock.
    vclk_be: vclk@be {
        compatible = "vrpc";
        server;
        info       = "vclk_ctrl";

    // Extra RPC channel back-end to execute enable/disable in interrupt context.
    vclk_fast_be: vclk_fast@be {
        compatible = "vrpc";
        server;
        info       = "vclk_fast_ctrl";
    };
};

&vm3_vdevs {

    // Default RPC channel front-end for virtual clock.
    vclk@fe {
        peer-phandle = <&vclk_be>;
        client;
        info         = "vclk_ctrl";
    };

    // Extra RPC channel front-end to execute enable/disable in interrupt context.
    vclk_fast@fe {
        peer-phandle = <&vclk_fast_be>;
        client;
        info         = "vclk_fast_ctrl";
    };
};


// VM2 Linux DTS configuring back-end.

    // clock provider A
    my_clk_providerA: pllA {
        compatible = "vendor,some-clk-controllerA";
        #clock-cells = <2>;
    };

    // clock provider B
    my_clk_providerB: pllB {
        compatible = "vendor,some-clk-controllerB";
        #clock-cells = <1>;
    };

    // clock consumer: disabled but kept for parsing.
    my_clk_consumer: devx {
        compatible = "disabled";
        clocks = <&my_clk_providerA 433 26>, <&my_clk_providerB 132>;
        clock-names = "mainclk", "auxclk";
    };

    /*
      Node describing clock signals that are exported to VM3.
      Device tree may contain multiple "vl,vclk-be" compatible nodes,
      one per peer VM.
    */
    vlx-clk-be@3 {

               compatible = "vl,vclk-be";

               // This node describes clocks exported to VM3.
               vl,vm-id = <3>;

               // We export two clk providers to VM3.
               vl,clock-providers = <&my_clk_providerA>, <&my_clk_providerB>;

               // Names identifying the providers. Those will be used by
               // vclk front-end in VM3.
               vl,clock-provider-names = "my_clk_providerA", "my_clk_providerB";
```

```
                // We only export the signals used by one or several clock
                // consumer.
                clk@0 {
                    vl,clock-ref = <&my_clk_consumer>;

                    // We use the extra RPC channel to execute enable/disable
                    // on the back-end in interrupt context.
                    vl,use-fast-rpc;
                };

    };
```

1.2.9.6 SEE ALSO

vrpc

`Device tree bindings for clock providers and clock consumers.`

1.2.10 VETH(4D)

1.2.10.1 NAME

veth — Virtual Ethernet Device driver interface

1.2.10.2 SYNOPSIS

#include <nk/nkern.h>

The virtual Ethernet driver implements a communication link between two VMs using an Ethernet like interface.

1.2.10.3 FEATURES

The virtual ethernet driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> VLX Virtual Ethernet driver

It can be compiled as a loadable module or as an embedded driver.

Virtual ethernet devices must be declared in the platform device tree. The example below declares veth devices using the virtual link framework(see the vlink_lookup manual page for more details).

```
&vm2_vdevs {
        vnet_23_0: vnet@23_0 {     // vnetwork VM2 <-> VM3
        compatible = "veth2";      // uses veth v2 protocol
        info = ",vnet23_0,,,16";   // NIC name, NAPI enabled (weight 16)
    };                             // server & client end points
}

vm3_vdevs {
        vnet@32_0 {                          // vnetwork VM2 <-> VM3
        peer-phandle = <&vnet_23_0>;     // peer vLINK
        info = ",vnet32_0,wakeup=250";   // NIC name, wakeup time-out 250 msecs, NAPI disabled
    };                                   // server & client end points
}
```

The 'info' property can be used.

The syntax is: [<address>][,[<name>][,[wakeup=<wakeup>][,[luoio][,[<napi>]]]]]

- <address> : [optional], the mac address in the form xx:xx:xx:xx:xx:xx
- <name> : [optional], the name of the network interface
- <wakeup> : [optional], wave lock timeout in milliseconds
- luoio : [optional], if present, the link is managed with the "Link-Up Only If Open" policy
- <napi> : [optional], the NAPI weight in [0, 2^31[. The value '0' disables the NAPI, any other value enables the NAPI with the weight set to this value

1.2.10.4 BRIDGE DRIVER

The underlying bridge driver used to communicate with both frontend and backend drivers is based upon the Hypervisor P2P communication link bridge driver described in vlink_lookup manual page.

1.2.10.5 DESCRIPTION

The virtual Ethernet driver can either run on the same or other VM. Both frontend and backend drivers act as a pair: a single frontend is connected to a single backend.

Two separated links are used in order to communicate between frontend and backend drivers: a virtual link for "transmit" ring buffers and the other one for "receive" ring buffers. In other words, both "transmit" and "receive" rings are controlled by a dedicated *NkDevVlink* object whose fields are explained in vlink_lookup manual page.

Consequently, the transmit queue of the frontend driver is connected to the receive queue of the backend driver and conversely. A new virtual Ethernet device is created if both virtual links are found from the Hypervisor repository. The virtual link used for receive ring buffers is found when the **s_id** field belonging to this *NkDevVlink* object is equal to the current guest OS identifier (returned by **nk_id_get(3D)**). The virtual link used for transmit ring buffers is found when the fields **s_id** and **c_id** of this another *NkDevVlink* object are respectively equal to the **c_id** and **s_id** fields of the *NkDevVlink* associated with the "receive" ring buffers.

If all these conditions are met, a new virtual Ethernet device is created. Both "receive" and "transmit" ring buffers are allocated in the Hypervisor repository through nk_pmem_alloc using as parameter *NkResourceId* a predefined identifier *VETH_PMEM_ID* (set to 4 by default).

A cross interrupt is allocated for receive operations through nk_pxirq_alloc and using as parameter *NkResourceId* a predefined *VETH_RXIRQ_ID* (set to 6 by default)&. Similarly, a cross interrupt is allocated for transmit operations with also a predefined *VETH_TXIRQ_ID* value (set to 7 by default). A handler for receiving frames is hooked to the receive cross interrupt **veth_rx_handler()**, and a handler invoked when the virtual link for transmit operations is ready is hooked to the transmit cross interrupt **veth_tx_ready_hdl()**.

When all resources are allocated for each virtual link, a *NK_XIRQ_SYSCONF* event is sent through **veth_sysconf_trigger()** primitive to the peer driver in order to start up the handshake protocol (see vlink_lookup manual page for a full description of this protocol).

The following API is exported by the virtual Ethernet driver running on top of Linux. The following sections are devoted to these primitives:

- Open a virtual Ethernet device

- Close a virtual Ethernet device
- Start a transmit operation
- Transmit timeout error handler
- Receive operations handler
- Transmit ready handler

On top of Linux a *struct net_device* object is allocated per virtual device (see **include/linux/netdevice.h** file for further details), and a private data object *VEth* is also allocated whose fields are shown below:

```
/*
 * Device instance data.
 */
typedef struct VEth {
    struct net_device_stats stats;      /* Network statistics     */
    struct net_device*      netdev;     /* Linux network device   */
    VEthLink                link;       /* Link with peer OS data */
} VEth;
```

Each primitive mentioned above are invoked with a *struct net_device* parameter and its corresponding *VEth* object. The **netdev** field allows to retrieve the associated *struct net_device* object for a given *VEth* object (that is, a back link pointer). A *VEthLink* is a local object used on both frontend and backend side for each connection between the current VM (that is, where the driver is running) to its peer VM (that is, where either the frontend or the backend is running):

```
typedef struct VEthLink {
    NkDevVlink*    rx_link;      /* RX vlink */
    VEthRingDesc* rx_ring;      /* RX ring */
    nku8_f*        rx_data;      /* RX persistent shared memory */
    nku32_f        max;
    nku32_f        min;
    nku32_f        sum;
    NkDevVlink*    tx_link;      /* TX vlink */
    VEthRingDesc* tx_ring;      /* TX ring */
    nku8_f*        tx_data;      /* TX persistent shared memory */
    struct VEth*   veth;
    VEthLocal      local;    /* For local cross interrupt management */
    VEthPeer       peer;     /* For remote cross interrupt management */
    int            enabled;
} VEthLink;
```

As previously mentioned, a couple of *NkDevVlink* objects are used, one for receiving operations and the other for transmit operations. The couple of objects *VEthLocal* and *VEthPeer* are respectively used in order to manage local cross interrupt: receive operation handler cross interrupt and its corresponding identifier, and to manage remote cross interrupt: transmit ready operation cross interrupt and its corresponding identifier:

```
typedef struct {
    NkOsId      osid;           /* rx_ring->s_id = current OS identifier */
    NkXIrq      rx_xirq;        /* store rx xirq number */
    NkXIrqId    rx_xid;         /* rx xirq handler id */
    NkXIrq      tx_ready_xirq;  /* store tx_ready xirq number */
    NkXIrqId    tx_ready_xid;   /* tx_ready xirq handler id */
} VEthLocal;

typedef struct {
    NkOsId      osid;           /* tx_ring->s_id = peer OS identifier */
    NkXIrq      rx_xirq;        /* xirq to send to peer OS */
    NkXIrq      tx_ready_xirq;  /* xirq to send to peer OS */
} VEthPeer;
```

All these fields are updated (cross interrupts are allocated and all appropriate handlers are attached as mentioned above) when both virtual links for receive and transmit operations are detected. Finally, a couple of ring descriptors *VEthRingDesc* objects are also allocated per virtual link whose fields are described below:

```
typedef struct VEthRingDesc {
    volatile nku32_f p_idx;     /* Producer index */
    volatile nku32_f freed_idx; /* freed slot index */
    volatile nku32_f c_idx;     /* Consumer index */
    volatile nku8_f  stopped;   /* Reflect dev\&. state (started=0/stopped=1) */
    nku16_f          size;      /* Size of the ring (number of slots) */
} VEthRingDesc;
```

The communication relies on a data ring with RING_SIZE slots (set to 64 per default). The ring descriptor and the data slots are both allocated in the same shared memory segment (that is, the Hypervisor repository). For performance purposes, data copied in the ring must be aligned on cache line boundaries. Therefore each slot, IP header and shared info structure are aligned. It is also possible to use a DMA for avoiding copies.

1.2.10.6 OPEN A VIRTUAL ETHERNET DEVICE

1.2.10.6.1 SYNOPSIS

This primitive is invoked to open a virtual Ethernet device.

#include <nk/nkern.h>

int **veth_ndo_open** (struct net_device∗ *dev*);

1.2.10.6.2 DESCRIPTION

This primitive triggers a **NK_XIRQ_SYSCONF** cross interrupt and starts the Ethernet device by invoking the Linux kernel **netif_start_queue()** primitive.

1.2.10.6.3 PARAMETERS

The single *dev* parameter is a valid pointer to a *struct net_device* object.

1.2.10.6.4 RETURN VALUES

This primitive never fails, a null value is always returned.

1.2.10.7 CLOSE A VIRTUAL ETHERNET DEVICE

1.2.10.7.1 SYNOPSIS

This primitive is invoked to close a previously virtual Ethernet device opened with **veth_ndo_open()**.

#include <nk/nkern.h>

int **veth_ndo_close** (struct net_device∗ *dev*);

1.2.10.7.2 DESCRIPTION

This primitive triggers a **NK_XIRQ_SYSCONF** cross interrupt and stops the Ethernet device by invoking the Linux kernel **netif_stop_queue()** primitive.

1.2.10.7.3 PARAMETERS

The single *dev* parameter is a valid pointer to a *struct net_device* object.

1.2.10.7.4 RETURN VALUES

This primitive never fails, a null value is always returned.

1.2.10.8 START A TRANSMIT OPERATION

1.2.10.8.1 SYNOPSIS

This primitive is called for starting a transmit operation.

#include <nk/nkern.h>

int **veth_ndo_start_xmit** (struct sk_buff∗ *skb*, struct net_device∗ *dev*);

1.2.10.8.2 DESCRIPTION

This primitive checks the current state of the virtual link. If it is not ready (that is, different from *NK_DEV_VLINK*←*_ON*), the transmit error carrier lost counter is incremented (that is, **tx_carrier_errors** field of the associated *struct net_device_stats* object). If the ring is full, the transmit fifo error counter is incremented and an error is returned. If the transmit ring is not full, the frame is copied into the appropriate ring buffer through **ring_push_data()** primitive, and a cross interrupt is sent to the peer driver and processed by the remote **veth_rx_hdl()** receive handler (see RECEIVE OPERATIONS HANDLER section for further details). If the ring is full after filling the ring buffer, this ring is temporarily stopped, and the associated queue is also stopped (that is, the **netif_stop_queue()** Linux kernel primitive is invoked). The transmit ring will be unblocked, when a cross interrupt will be received for either a transmit ready event, or a transmit timeout error event.

1.2.10.8.3 PARAMETERS

The first *skb* parameter is a valid pointer to a *struct sk_buff* object whose fields are described in **include/linux/skbuff.h** file. The second *dev* parameter is a valid pointer to a *struct net_device* object.

1.2.10.8.4 RETURN VALUES

In case of success, *NETDEV_TX_OK* is returned, otherwise *NETDEV_TX_BUSY* is returned when the transmit ring is full. In that case, the transmit fifo error counter is incremented (that is, **tx_fifo_errors** field of the associated *struct net_device_stats* object), and the frame is dropped. Note that *NETDEV_TX_OK* is also returned when the virtual link is down. In that case, the transmit error carrier lost counter is incremented, and the frame is dropped.

1.2.10.9 TRANSMIT ERROR TIMEOUT HANDLER

1.2.10.9.1 SYNOPSIS

This handler is invoked upon expiration of a watchdog timer.

#include <nk/nkern.h>

void **veth_ndo_tx_timeout** (struct net_device∗ *dev*);

1.2.10.9.2 DESCRIPTION

This handler is armed upon initialization, and is used as a watchdog mechanism for the transmit operations. If the transmit queue is full, a cross interrupt is sent to its peer driver in order to trigger the receive handler (that is, **veth_rx_hdl()** handler, see the following section for further details). If the transmit queue is not full, the queue is restarted by calling the Linux kernel primitive **netif_wake_queue()**, and its current state is no longer stopped.

1.2.10.9.3 PARAMETERS

The single *dev* parameter is a valid pointer to a *struct net_device* object.

1.2.10.9.4 RETURN VALUES

No relevant value is returned from this handler.

1.2.10.10 RECEIVE OPERATIONS HANDLER

1.2.10.10.1 SYNOPSIS

This handler is invoked when data or error is detected on the receive ring.

#include <nk/nkdev.h>

void **veth_rx_xirq** (void∗ *cookie*, NkXirq *xirq*);

1.2.10.10.2 DESCRIPTION

This handler is invoked when a frame is successfully sent by the peer driver, or when a timeout has occurred on the peer transmit ring by triggering **veth_tx_timeout()** (see TRANSMIT ERROR TIMEOUT HANDLER section for further details). If the receive ring is not empty, and the virtual link is active (that is, current state set to *NK_D*←*EV_VLINK_ON*), a new buffer is allocated and is used to get the received frame from the ring (see the couple of primitives: **ring_pull_data()** and **veth_alloc_skb()** for further details). Then Ethernet statistics are updated and if the current state of the receive queue is set to stopped, a cross interrupt is sent to the peer driver in order to invoked **veth_tx_ready_xirq()** handler (see TRANSMIT QUEUE READY HANDLER section for further details, and also **veth_free_skb()** primitive which is used to trigger this interrupt when flag -DCONFIG_SKB_DESTRUCTOR is turned on).

The interface reception operates in one of the two modes:

- normal IRQ mode: each RX IRQ is taken and processed individually,
- NAPI mode: the first RX IRQ is taken and it disables the subsequent ones, a certain amount (i.e. budget) of the subsequent RX IRQs is processed in polling mode, then, when no more RX IRQ is available, RX IRQs are enabled again. The mode is selected in accordance to the <napi> parameter in the vlink description.

1.2.10.10.3 PARAMETERS

The first *cookie* parameter is a valid pointer on a *VEthLink* in the context of this handler, allowing to quickly retrieve the transmit ring descriptor (that is, *VEthRingDesc* object), and the *VEth* associated object. The second parameter is the value of the cross interrupt and is not used by this handler.

1.2.10.10.4 RETURN VALUES

No relevant value is returned from this handler.

1.2.10.11 TRANSMIT QUEUE READY HANDLER

1.2.10.11.1 SYNOPSIS

This handler is invoked when the transmit ring can accept data.

#include <nk/nkern.h>

void **veth_tx_ready_xirq** (void∗ *cookie*, NkXirq *xirq*);

1.2.10.11.2 DESCRIPTION

This handler is invoked when the transmit ring is ready to accept a new frame. This activation is done by the peer **veth_rx_hdl()** handler when the transmit ring state has been set to stopped upon start transmit operation (that is, local **veth_start_xmit()**), and detecting a transmit ring full after getting the last buffer ring. This handler is also triggered when a buffer is freed (that is, a *struct sk_buff* object freed by **veth_free_skb()**). Both statements are similar, the former is processed when the -DCONFIG_SKB_DESTRUCTOR option is not set, and the latter when this option is set.

1.2.10.11.3 PARAMETERS

The first *cookie* parameter is a valid pointer on a *VEthLink* in the context of this handler, allowing to quickly retrieve the transmit ring descriptor (that is, *VEthRingDesc* object), and the *VEth* associated object. The second parameter is the value of the cross interrupt and is not used by this handler.

1.2.10.11.4 RETURN VALUES

No relevant value is returned from this handler.

1.2.10.12 SEE ALSO

nk_id_get

nk_mem_map

nk_pmem_alloc

nk_ptov

nk_pxirq_alloc

nk_vtop

nk_vlink_lookup

nk_xirq_attach

nk_xirq_detach

nk_xirq_trigger

1.2.11 VEVENT_BE(4D)

1.2.11.1 NAME

vevent_be — Virtual event back-end driver interface

1.2.11.2 SYNOPSIS

#include <vlx/vevdev_common.h>

The virtual event backend is used to intercept mouse, keyboard or touch screen input events and forward them to a frontend driver. Multiple frontend drivers can be connected to a single backend driver managing a mouse, a keyboard or a touch screen.

The virtual event backend driver uses the Linux input device framework to find corresponding physical devices and intercept input events from them. These physical devices will be hidden from the user land application after vevent driver initialization. Instead of them, vevent frontend driver will export corresponding virtual devices on the same or another VM.

1.2.11.3 FEATURES

The virtual event backend driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Vevent backend interface

It can be compiled as a loadable module or as an embedded driver.

Virtual event devices must be declared in the platform device tree. The example below declares vevent devices using the virtual link framework(see the vlink_lookup manual page for more details).

```
&vm2_vdevs {
   vevent_0: vevent@be.0 {
        compatible = "vevent";
        #clone     = "auto";
        server;
        info =
             "bootargs:",
             // <boolean-expression> ":" <topology> ":" <policy> ";" |
             //                      ":" <topology> ":;"
             // physical power button:
             // (!BUS_VIRTUAL && EV_KEY && KEY_POWER && !KEY_Q)
             "idev_bustype!=(u16)6&&",
             "(idev_evbit[0]&(u32)0x2)!=(u32)0&&",
             "(idev_keybit[3]&(u32)0x100000)!=(u32)0&&",
             "(idev_keybit[0]&(u32)0x100)==(u32)0:",
             "2,3:DEFAULT-focus;",
    };
}

&vm3_vdevs {
    vevent@fe.0 {
        peer-phandle = <&vevent_0>;
        client;
    };
}
```

The *info* property can be used to add backend parameters. In particular the *bootargs* prefix defines the input devices configuration. The vevent driver default setup syntax is defined in details in the following section VEVENT SETUP SYNTAX.

1.2.11.4 VIRTUAL LINKS

The virtual event frontend driver is based on the virtual link framework described in vlink_lookup manual page.

1.2.11.5 DESCRIPTION

The virtual event backend driver provides an API to process keyboard or touch screen events to frontend drivers running on the same or other VM. The backend driver is responsible for accessing the physical keyboard or touch screen invoking an underlying native driver.

1.2.11.6 EXTENDED DESCRIPTION

The virtual event device is an abstraction which enables VM to access either a keyboard or a touch screen devices managed by VM. It exports standard Linux API for the input devices.

The virtual event backend driver uses the NKDDI to provide communications and synchronization between frontend driver(s) and the backend driver. It uses standard Linux API for the input devices in order to intercept input events from the real input devices and forward them to frontend drivers.

The virtual event backend driver exports the objects *VRing* and *VIdev* in the Hypervisor repository (shared persistent memory):

```
#define VRING_SIZE              256
#define VRING_MASK              (VRING_SIZE - 1)
#define VRING_POS(idx)          ((idx) & VRING_MASK)
#define VRING_PTR(rng,idx)      (rng)->ring[VRING_POS((rng)->idx)]
#define VRING_C_ROOM(rng)       ((rng)->p_idx - (rng)->c_idx)
#define VRING_C_CROOM(rng)      (VRING_SIZE - VRING_POS ((rng)->c_idx))
#define VRING_P_ROOM(rng)       (VRING_SIZE - ((rng)->p_idx - (rng)->c_idx))

typedef struct {
        nku16_f type;
        nku16_f code;
        nku32_f value;
} input_vevent ;

typedef struct VRing {
        nku32_f      c_idx;            /* "Free running" consumer index */
        nku32_f      p_idx;            /* "Free running" producer index */
        input_vevent ring[VRING_SIZE]; /* Circular communication buffer */
} VRing;

                             // Linux 2.6.9
#define VEVENT_EV_CNT  0x20  // 0x20
#define VEVENT_KEY_CNT 0x600 // 0x300
#define VEVENT_REL_CNT 0x20  // 0x10
#define VEVENT_ABS_CNT 0x80  // 0x40
#define VEVENT_MSC_CNT 0x10  // 0x08
#define VEVENT_LED_CNT 0x20  // 0x10
#define VEVENT_SND_CNT 0x10  // 0x08
#define VEVENT_FF_CNT  0x100 // 0x80
#define VEVENT_SW_CNT  0x21  // 0x10

#define VEVENT_INPUT_PROP_CNT   0x20

#define BITS_TO_INTS(x)  (((x) + (sizeof(int) * 8) - 1) / (sizeof(int) * 8))

typedef struct {
    // Currently we only support following combined events:
    // More information see "struct input_dev" in linux kernel header include/linux/input.h
    unsigned int evbit[BITS_TO_INTS(VEVENT_EV_CNT)];
    unsigned int keybit[BITS_TO_INTS(VEVENT_KEY_CNT)];
```

```
    unsigned int absbit[BITS_TO_INTS(VEVENT_ABS_CNT)];
    unsigned int relbit[BITS_TO_INTS(VEVENT_REL_CNT)];
    unsigned int mscbit[BITS_TO_INTS(VEVENT_MSC_CNT)];
    unsigned int ledbit[BITS_TO_INTS(VEVENT_LED_CNT)];
    unsigned int sndbit[BITS_TO_INTS(VEVENT_SND_CNT)];
    unsigned int ffbit[BITS_TO_INTS(VEVENT_FF_CNT)];
    unsigned int swbit[BITS_TO_INTS(VEVENT_SW_CNT)];

    unsigned int key[BITS_TO_INTS(VEVENT_KEY_CNT)];
    unsigned int led[BITS_TO_INTS(VEVENT_LED_CNT)];
    unsigned int snd[BITS_TO_INTS(VEVENT_SND_CNT)];
    unsigned int sw[BITS_TO_INTS(VEVENT_SW_CNT)];

    int abs[VEVENT_ABS_CNT];
    int absmax[VEVENT_ABS_CNT];
    int absmin[VEVENT_ABS_CNT];
    int absfuzz[VEVENT_ABS_CNT];
    int absflat[VEVENT_ABS_CNT];

    unsigned int propbit[BITS_TO_INTS(VEVENT_INPUT_PROP_CNT)];
} VIdev;
```

The first object allocated by the backend in the repository is a *VRing* followed by a *VIdev*.

The fields **c_idx** and **p_idx** are respectively the current consumer index updated by the frontend driver when a cross interrupt signaling an input event is sent by the backend driver, and the producer index updated by the backend driver in **evdev_event** primitive. The *ring* is composed of *VRING_SIZE* events used as a circular buffer. The free indexes are managed through the couple of macros *VRING_C_ROOM* and *VRING_C_CROOM* on the frontend side, and through the *VRING_P_ROOM* macro on the backend side.

The *VIdev* contains the current virtual device configuration updated at initialization time. The *absbit[BITS_TO_INTS(VEVENT_ABS_CNT)]* is a bitmap used to record at initialization time the various events associated with either keyboard/keypad or touch screen device: X axis, Y axis, absolute pressure, and so on (all these values are fully described in **include/linux/input.h file**, see *ABS_XXX*). The maximum values for events coming from absolute axes are contained in the *absmax[VEVENT_ABS_CNT]* table. The minimum values for events coming from absolute axes are contained in the *absmin[VEVENT_ABS_CNT]* table. The *absfuzz[VEVENT_ABS_CNT]* table holds noisiness values for axes. The *absflat[VEVENT_ABS_CNT]* table holds the size of the center flat position (used only by joydevices touch screen.

On top of Linux the virtual backend driver exports the following primitives through *struct input_handler* tables for each device (see also **linux/include/input.h** file for further details):

vevdev_ih_filter

process an event coming from an input device

vevdev_ih_connect

connect an input device

vevdev_ih_disconnect

disconnect an input device

1.2.11.7 CONNECTING AN INPUT DEVICE

1.2.11.7.1 SYNOPSIS

Prior accessing an input device, the primitive **connect** will be called by the Linux input device framework.

#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/input.h>
#include <linux/major.h>
#include <linux/device.h>
#include <nk/nkern.h>
#include <vlx/vevdev_common.h>

void **vevdev_ih_connect** (struct input_handler∗ *handler*, struct input_dev∗ *dev*, const struct input_device_id∗ *id*);

1.2.11.7.2 DESCRIPTION

This primitive allocates the appropriate underlying data structures to manage the input device. The following objects are allocated or retrieved (that is, reserved) from the Hypervisor repository (shared persistent memory between backend and frontend drivers):

- A *device_t* object is explicitly allocated
- A *frontend_t* object per virtual link having *vevent* as name, the correct logical virtual link number and server identifier (that is, VM identifier) is allocated
- According to the same conditions described above, the objects *VRing* and *VIdev* are allocated/reserved through the vlink_lookup and nk_pdev_alloc primitives from the Hypervisor repository.

All these data structures are initialized and cross interrupts are also allocated to communicate with the frontend driver(s). Finally, the virtual link protocol handshake is started up in order to enable communication between a backend and all its peer counterpart (that is, frontend driver(s)). The vlink_lookup manual page can be consulted for further details about this handshake protocol.

A *device_t* object is defined as follows in the backend driver:

```
typedef struct device_t {
    struct list_head list;
    struct input_handle handle;
    topology_t topology;
    topology_t fe_topology;
    policy_t policy;
    state_t state;
} device_t;
```

See section VEVENT SETUP SYNTAX for more details about *topology* and *policy* fields.

A *frontend_t* object is defined as follows in the backend driver:

```
typedef struct {
    NkDevVlink* vlink;       /* consumer/producer link */
    VRing*      vring;       /* consumer/producer circular ring */
    NkXIrq      xirq;        /* cross interrupt to send to consumer OS */
    VIdev*      videv;       /* input device configuration */
    vevdev_identity_t *identity;
    NkPhAddr    pdata;       /* shared memory physical address */
    _Bool       link_up_allowed;
} frontend_t;
```

These data structures are used internally to record pointers on objects exported by the backend driver.

1.2.11.7.3 PARAMETERS

The first argument is a valid pointer on a *struct input_handler* object. The second argument is a valid pointer to a *struct input_dev* object. The third argument is a valid pointer to a *struct input_device_id*. (see also **linux/include/input.h** for further details about this three arguments).

1.2.11.7.4 RETURN VALUES

This primitive returns 0 in case of success. The following error codes are returned in case of error:

-ENOMEM

If there is not enough memory to allocate underlying objects.

1.2.11.8 PROCESSING AN INPUT EVENT

1.2.11.8.1 SYNOPSIS

All input events provided are processed with the **filter** primitive.

```
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/input.h>
#include <linux/major.h>
#include <linux/device.h>
#include <nk/nkern.h>
#include <vlx/vevdev_common.h>
```

bool **vevdev_ih_filter** (struct input_handle∗ *handle*, unsigned int *type*, unsigned int *code*, unsigned int *value*);

1.2.11.8.2 DESCRIPTION

This primitive is responsible for intercepting all events generated by input devices and forward them to frontend drivers. If the current state of the virtual link is alive (that is, current state is equal to *NK_DEV_VLINK_ON*), a cross interrupt is sent to the appropriate frontend driver. If the virtual link is off, the event is discarded.

1.2.11.8.3 PARAMETERS

The first argument is a valid pointer to a *struct input_handle* object. The remaining arguments are respectively an event type, an event code and a value of an event. This event is stored in the virtual ring (*input_event* object) accessible by both backend and frontend drivers (see **linux/include/input.h** file for further details about data structures and argument values).

1.2.11.8.4 RETURN VALUES

This handler returns the TRUE boolean value if the event has been filtered.

1.2.11.9 DISCONNECTING AN INPUT DEVICE

1.2.11.9.1 SYNOPSIS

The **disconnect** is invoked by the Linux input device framework to disconnect an input device.

#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/input.h>
#include <linux/major.h>
#include <linux/device.h>
#include <nk/nkern.h>
#include <vlx/vevdev_common.h>

void **vevdev_ih_disconnect** (struct input_handle∗ *handle*);

1.2.11.9.2 DESCRIPTION

This primitive releases a virtual keyboard or touch screen device and deletes all underlying objects previously allocated by **vevdev_ih_connect**.

1.2.11.9.3 PARAMETERS

The single argument is a valid pointer to a *struct input_handle* object.

1.2.11.9.4 RETURN VALUES

This primitive does not return any error code.

1.2.11.10 SEE ALSO

nk_id_get

nk_pdev_alloc

nk_ptov

vlink_lookup

nk_vtop

nk_xirq_attach

nk_xirq_detach

nk_xirq_trigger

vevent_fe

1.2.11.11 VEVENT SETUP SYNTAX

This section aims at describing the syntax involved in vevent driver configuration.

1.2.11.11.1 Overview

Vevent is the virtualization of the Linux kernel input framework (see Documentation/input/... in Linux kernel for details). It is in charge of receiving input events from the physical input devices, and delivering them transparently to the "appropriate" domains through vlm virtualization system. In this context "appropriate" means in accordance with vevent input device virtualization setup. The next sections focus on how to implement such a setup.

Physical input devices register and unregister in the input framework at anytime. Platform input device register at system boot time, whereas USB and BT input devices (mice, Keypad, ...) may register when the system is already running.

When a new physical input device registers in the system, vevent checks if a default virtualization setup exists for this device in the database, and applies this default setup if so. If no default setup is found, the device remains pending for a runtime setup to be provided from userland. Note that vevent runtime setup feature is not implemented at this time. The virtualization setup for an input device includes 2 parameters:

- topology
- policy

These parameters are described in the next sections.

1.2.11.11.2 Setup Parameter: topology

This parameter indicates which are the possible recipient domains (i.e. VMs) for the events sent by the device. This is used to create the appropriate frontend instances accross the VMs. A VM which is part of the topology may receive events from the device in accordance with its policy. A VM which is not part of the topology will never receive any events from the device. A possible recipient domain is identified in the topology by the number of its associated VM. A special "native" value for topology explicitly indicates that the device is not virtualized, and that it is managed natively by the kernel. The policy value is meaningless for such devices.

1.2.11.11.3 Setup Parameter: policy

This parameter indicates on a per event type basis, the routing method to be used to convey events from the device to the recipient domain(s) of the topology.

Valid routing methods are:

- broadcast: events are broadcast to –all– recipients which are part of the device topology.
- drop: events are simply discarded.
- focus: events are forwarded to the only –one– recipient domain which is currently focused.
- fixed: events are forwarded to the listed domains of the topology.

Valid event types are defined in the Linux kernel (see **include/linux/input.h file**):

- EV_KEY event types are related to key presses (keypads),
- EV_ABS event types are related to absolute coordinate devices (touchpads),
- EV_REL event types are related to relative coordinate devices (mice),
- ...

Defining a policy for an input device basically comes to chose a routing method for each of the supported event types. To make things easier, a builtin "DEFAULT" event type acts as a fallback for all event types which have not been specifically assigned a routing method. For most of usual devices, simply defining the DEFAULT method is sufficient.

Policy may be ommited, in that case the assumed policy is equivalent to: "DEFAULT-drop".

1.2.11.11.4 Input Device Type Recognition

Platform (builtin) devices can be most of the time recognized with their device name, which is known at setup time (and even before). Dynamic devices (USB/BT) recognition cannot rely on the device name, because it is not guaranteed to exist and it is likely to change from one device brand to another. In that conditions it is preferable for the device recognition process to rely on device capabilities (does it have keys events, does it have ABS support, ...) rather than on device names.

Vevent recognizes input devices analysing their descriptors (namely their "struct input_dev" data structure - see **include/linux/input.h** file). Basically, vevent database includes a set of boolean expressions combining various fields of the device descriptor. Each of these boolean expressions is evaluated for any new registered device, if the expression evaluates to "true", the device is considered as recognized and the associated default setup applied to it. If it evaluates to false, the next boolean expression in the database is considered. The process stops on the first found default setup, or when all expressions have been evaluated for a device.

Below are the fields of the "struct input_dev" which can be combined to form the boolean expression used in input device recognition process:

- name: character string,
- phys: character string,
- uniq: character string,
- id: array of 4 16 bit integers,
- id.bustype: 16 bit integer (same as id[0])
- id.vendor: 16 bit integer (same as id[1])
- id.product: 16 bit integer (same as id[2])
- id.version: 16 bit integer (same as id[3])
- propbit: array of 32 bit integers,
- evbit: array of 32 bit integers,
- keybit: array of 32 bit integers,
- relbit: array of 32 bit integers,
- absbit: array of 32 bit integers,
- mscbit: array of 32 bit integers,
- ledbit: array of 32 bit integers,
- sndbit: array of 32 bit integers,
- ffbit: array of 32 bit integers,
- swbit: array of 32 bit integers. The way fields can be combined in a boolean expression is described in section Boolean Expression Grammar.

1.2.11.11.5 Generic Setup vs. Platform Specific Setup

Input device setup is split in 2 files:

- a generic file, common to all platforms (./bsp/vdt/true/vevent.dtsi)
- a platform specific file (./bsp/board/xxx/xxx/vdt-true-vevent.dtsi)

At boot time, vevent driver merges these 2 files in the database. Platform specific setup are inserted before generic ones, so that they always have precedence in the lookup process. Generic setup file integrates some of the most usual input devices (power-on keys, keypad, mouse, touchscreen, headset) in such a way that moving to a new platform requires very little work.

1.2.11.11.6 Redbend Internal Input Devices

Most of input devices are implemented in Linux kernel, and their "struct input_dev" is available to our customers. There are a few particular Redbend input devices though, whose source code (and hence descriptor) is not delivered to our customers. These devices are recognized with their name in the generic setup: "vpower-button-vm2" and "vpower-button-vm3" are "logical" (in the sense there is no hardware associated with them) input devices, dynamically created by vpowerd daemon through /dev/uinput userland framework, and which implement a virtual power button for VM#2 and VM#3 respectively.

1.2.11.11.7 Overall Syntax

Default setup is a "big" character string, integrated in the vlink "info" field of the vevent vlink #0. It is made-up 4 sections: The first 3 ones define couples of name:value for device, topology and policy definition respectively. The last one assembles a device, a topology and a policy definition in a record of the setup database. Below the BNF description of the default setup grammar:

```
/* ============================================================ *
 * default setup
 * ============================================================ */
<default-setup> ::=
    <device-list>

<device-list> ::=
    <device> ";" <device-list> |
    <device>

<device> ::=
        <boolean-expr> ":" <topology> ":" <policy>

/* ============================================================ *
 * topology
 * ============================================================ */
<topology> ::=
    "native" |
    <vm-id-list>

/* ============================================================ *
 * policy
 * ============================================================ */
<policy> ::=
    <policy-event-list> |
    ""

<policy-event-list> ::=
                 <policy-event> |
                 <policy-event> "|" <policy-event-list>

<policy-event> ::=
            <event-type> "-" <policy-spec>

<event-type> ::=
           "KEY" |
           "REL" |
           "ABS" |
           "MSC" |
           "SW"  |
           "LED" |
           "SND" |
           "REP" |
           "FF"  |
           "PWR" |
           "FF_STATUS" |
           "DEFAULT"

<policy-spec> ::=
       "broadcast" |
       "drop"      |
```

```
       "focus"      |
       <vm-id-list>

/* ============================================================ *
 * common rules
 * ============================================================ */
<vm-id-list> ::=
        <vm-id> |
        <vm-id> "," <vm-id-list>

<vm-id> ::=
      integer in [2, 31]

<identifier> ::=
           character string
```

1.2.11.11.8 Boolean Expression Grammar

This section describes in details the syntax for <boolean-expr>.

Below are listed the supported features:

- syntax inpired from C syntax,
- operator precedence is identical to C,
- basic builtin types:
  - "U16" : 16 bit unsigned integer,
  - "U32" : 32 bit unsigned integer,
  - "BOOL" : boolean,
  - "STR" : character strings,
- builtin "true" and "false" BOOL constants,
- Array of U16/U32 basic types.
- builtin "constants" that refer to the struct input_dev fields (see grammar below)
- arithmetic operators (operate on U16/U32, return same type as operands): -"+" : addition, -"-" : substraction,
- logical operators (operate on BOOL, return BOOL):
  - "&&" : logical AND,
  - "||" : logical OR,
  - "!" : logical NOT,
- bitwise operators (operate on U16/U32, return same type as operands):
  - "&" : bitwise AND,
  - "|" : bitwise OR,
  - "^" : bitwise XOR,
  - "∼" : bitwise NOT,
- comparison operators (operate on U16/U32, BOOL, STR, return BOOL)
  - "==" : equals,
  - "!=" : is not equal,
- index operators (operate on arrays of U16/U32, return the same type as operand)
  - [index] : returns the "index"th elem of the array,

- [index1 .. index2] : returns a sub-array of the array

- optionnally the "C like" comma operator,
- optionnally the "C like" ternary (?:) operator.

Constraints/limitations:

- numeric U16/U32 constant must be typed with the (u16) and (u32) prefix,
- operands involved in arithmetic, bitwise or comparison operations must have the same type,

Below the BNF description of the supported boolean expression grammar:

```
/* ============================================================ *
 * common rules
 * ============================================================ */
<number> ::=
    <hex-number> |
        <dec-number>

<number-list> ::=
     <number> |
         <number> "," <number-list>

<number-array> ::=
    "{" <number-list> "}"

<hex-number> ::=
    "0x" <hex-digit-list>

<hex-digit-list> ::=
    <hex-digit> <hex-digit-list> |
    <hex-digit>

<hex-digit> ::=
    <dec-digit> |
    "a" | ... | "f"

<dec-number> ::=
    <dec-digit-list>

<dec-digit-list> ::=
    <dec-digit> <dec-digit-list> |
    <dec-digit>

<dec-digit> ::=
    "0" | "1" | ... | "9"

/* ============================================================ *
 * U16 type rules
 * ============================================================ */
<builtin-u16-variable> ::=
    "idev_id"      |
    "idev_bustype" |
    "idev_vendor"  |
    "idev_product" |
    "idev_version"

<u16-variable> ::=
     <builtin-u16-variable>                      |
         <u16-variable> "[" <number> "]"              |
         <u16-variable> "[" <number> ".." <number> "]"

<u16-value> ::=
    "(u16)" <number>                             |
        "(u16)" <number-array>                       |
        <u16-value> "[" <number> "]"                 |
```

```
        <u16-value> "[" <number> ".." <number> "]"    |
        <u16-value> "&" <u16-value>                   |
        <u16-value> "|" <u16-value>                   |
        <u16-value> "^" <u16-value>                   |
        "~" <u16-value>                               |
        "(" <u16-value> ")"                           |
        <u16-variable> "=" <u16-value>                |
        <u16-variable>                                |
        <bool-value> "?" <u16-value> ":" <u16-value>

/* ============================================================ *
 * U32 type rules
 * ============================================================ */
<builtin-u32-variable> ::=
    "idev_propbit" |
    "idev_evbit"   |
    "idev_keybit"  |
    "idev_relbit"  |
    "idev_absbit"  |
    "idev_mscbit"  |
    "idev_ledbit"  |
    "idev_sndbit"  |
    "idev_ffbit"   |
    "idev_swbit"

<u32-variable> ::=
     <builtin-u32-variable>                       |
         <u32-variable> "[" <number> "]"              |
         <u32-variable> "[" <number> ".." <number> "]"

<u32-value> ::=
    "(u32)" <number>                              |
        "(u32)" <number-array>                        |
        <u32-value> "[" <number> "]"                  |
        <u32-value> "[" <number> ".." <number> "]"    |
        <u32-value> "&" <u32-value>                   |
        <u32-value> "|" <u32-value>                   |
        <u32-value> "^" <u32-value>                   |
        "~" <u32-value>                               |
        "(" <u32-value> ")"                           |
        <u32-variable> "=" <u32-value>                |
        <u32-variable>                                |
        <bool-value> "?" <u32-value> ":" <u32-value>

/* ============================================================ *
 * BOOL type rules
 * ============================================================ */
<bool-value> ::=
    "true"                                          |
        "false"                                         |
        <bool-value> "&&" <bool-value>                  |
        <bool-value> "||" <bool-value>                  |
        "!" <bool-value>                                |
        "(" <bool-value> ")"                            |
        <u16-value> "==" <u16-value>                    |
        <u16-value> "!=" <u16-value>                    |
        <u32-value> "==" <u32-value>                    |
        <u32-value> "!=" <u32-value>                    |
        <bool-value> "==" <bool-value>                  |
        <bool-value> "!=" <bool-value>                  |
        <str-value> "==" <str-value>                    |
        <str-value> "!=" <str-value>                    |
        <bool-value> "?" <bool-value> ":" <bool-value>

/* ============================================================ *
 * STR type rules
 * ============================================================ */
<builtin-str-variable> ::=
    "idev_name" |
        "idev_phys" |
        "idev_uniq"

<str-value> ::=
```

```
    double quoted character string                |
        <builtin-str-variable>                        |
        <bool-value> "?" <str-value> ":" <str-value>

/* ============================================================ *
 * valid BOOLEAN EXPRESSION
 * ============================================================ */
<boolean-expr> ::=
    <bool-value>
```

1.2.11.11.9 Examples

Important notices:

- all arithmetic constants (U16, U32, array of U16 and array of U32 must be appropriately prefixed (i.e. with "(u16)" or "(u32)").
- boolean expression is the only place where space characters are accepted.
- (u16) and (u32) variables/values cannot be mixed.

expression below are –valid– expressions:

```
"(idev_evbit[1] & (u32)0x12) == (u32)0x2"
"(idev_evbit[1..3] & (u32){0x12, 0xff00ff, 2}) == (u32){0x2, 0x5500aa, 2}"
"(idev_evbit[1..3] ^ idev_keybit[0..2]) != idev_swbit[2..4]"
"(idev_name == \"my_input_device\") && (idev_vendor != (u16)0x2345)"
"idev_id == (u16){0x12, 23, 76, 0}"
"!(idev_id == (u16){0x12, 23, 76, 0}) != false"
```

expression below are –invalid– expressions:

```
"(idev_evbit[1] & (u16)0x12) == (u32)0x2"
"idev_id != true"
```

Below a "C-like" short example of a syntactically valid string for a default setup (excerpt from hammerhead platform setup):

```
"idev_name==\"msm8974-taiko-mtp-snd-card Button Jack\":native :;"
"idev_name==\"msm8974-taiko-mtp-snd-card Headset Jack\":native :;"
// physical power button:
// (!BUS_VIRTUAL && EV_KEY && KEY_POWER && !KEY_Q)
"idev_bustype!=(u16)6&&(idev_evbit[0]&(u32)0x2)!=(u32)0&&"
"(idev_keybit[3]&(u32)0x100000)!=(u32)0&&(idev_keybit[0]&(u32)0x100)==(u32)0:"
"2,3:DEFAULT-focus;"
// headset:
// (EV_SW && (SW_HEADPHONE_INSERT || SW_MICROPHONE_INSERT))
"(idev_evbit[0]&(u32)0x20)!=(u32)0&&(idev_swbit[0]&(u32)0x14)!=(u32)0:"
"2,3:SW-broadcast|DEFAULT-focus;"
// virtual power buttons
"idev_name==\"vpower-button-vm2\":2:DEFAULT-2;"
"idev_name==\"vpower-button-vm3\":3:DEFAULT-3;"
// Keypad if device has at least one button
"(idev_evbit[0]&(u32)0x2)!=(u32)0&&idev_keybit[0..7]!=(u32){0,0,0,0,0,0,0,0}:"
"2,3:DEFAULT-focus;"
// Mouse:
// (EV_REL && REL_X && REL_Y)
"(idev_evbit[0]&(u32)0x4)!=(u32)0&&(idev_relbit[0]&(u32)0x3)==(u32)0x3:"
"2,3:DEFAULT-focus;"
// Touchscreen:
// (EV_ABS &&
//  ABS_MT_TOUCH_MAJOR && ABS_MT_TOUCH_MINOR &&
//  ABS_MT_POSITION_X && ABS_MT_POSITION_Y)
"(idev_evbit[0]&(u32)0x8)!=(u32)0&&(idev_absbit[1]&(u32)0x6618000)==(u32)0x6618000:"
"2,3:DEFAULT-focus;"
// Default setup:
// very convenient, BUT will prevent setup from
// userland when it will implemented.
"true:2,3:DEFAULT-focus;"
```

note that "" are escaped. note the last rule, which applies to all unknown devices.

- For an exhaustive example of the boolean expression syntax, please refer to host tests scenario file (vdriver-xxx/include/vlx/vevdev-expr-test.m).

1.2.12 VEVENT_FE(4D)

1.2.12.1 NAME

vevent_fe — Virtual Event front-end driver interface

1.2.12.2 SYNOPSIS

#include <vlx/vevdev_common.h>

The virtual event frontend driver is used in order to receive events from real keyboard and touch screen devices managed by a backend driver and forward them to corresponding virtual devices. Both backend and frontend drivers may run on the same VM.

The virtual event frontend driver is integrated in the Linux input device framework. It exports standard input devices like /dev/input/event∗ to the user land applications.

1.2.12.3 FEATURES

The virtual event frontend driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Vevent frontend interface

It can be compiled as a loadable module or as an embedded driver.

Virtual devices are declared in the platform device tree for each VM. An example is given in the vevent_be manual page.

1.2.12.4 VIRTUAL LINKS

The virtual event frontend driver is based on the virtual link framework described in vlink_lookup manual page.

1.2.12.5 DESCRIPTION

The virtual event device is an abstraction which enables a VM to access either a keyboard or a touch screen device managed by another VM. It exports standard Linux API for the input devices. The virtual event frontend driver uses the NKDDI in order to communicate with the backend driver. It uses standard Linux input devices framework in order to export virtual event devices to the user land.

1.2.12.6 EXTENDED DESCRIPTION

The virtual event frontend driver communicates with its peer counterpart (that is, backend driver) through cross interrupts and a repository (that is, persistent shared memory provided by the Hypervisor) containing a couple of objects *VRing* and *VIdev* described in detail in the vevent_be manual page. Each frontend driver also allocates a private *Vdev* having the following layout:

```
typedef struct {
    NkDevVlink* vlink;          /* consumer/producer link */
    VRing*      vring;          /* consumer/producer circular ring */
    NkXIrq      xirq;           /* cross interrupt number */
    vevdev_identity_t* identity;
    NkXIrqId    xid;            /* cross interrupt id */
    struct input_dev* idev;
    _Bool       registered;     /* input_dev registered already or not */
    const char* name;           /* from vlink s_info */
    VIdev*      videv;          /* input device configuration */
    int         mt_slot_max;    /* Multitouch max slot */
    struct work_struct create_work;
    struct work_struct delete_work;

    NkPhAddr           pdata;       /* shared memory physical address */

#ifdef VEVDEV_PROC_FS
    /* Statistics */
    struct {
    unsigned    total_events;
    unsigned    created;
    unsigned    deleted;
    unsigned    last_event;
    } stats;
#endif

#ifdef VEVDEV_MT
    /* Support for multi-touch EV_ABS devices */
    unsigned    mt_active_slots;
    unsigned    mt_current_slot;
#endif

#ifdef VEVDEV_HISTORY
    input_vevent  history [256];
    unsigned    history_next;
    unsigned    history_high;
#endif
} Vdev;
```

This data structure is used internally in order to record pointers on objects exported by the backend driver.

At initialization time, each frontend driver executes **vdev_init** allowing to get this couple of objects and initializing all these data structures. Finally, the handshake protocol with the backend driver is executed invoking **vdev_↩ handshake** (see the vlink_lookup manual page for further details about this protocol).

A cross interrupt handler **vdev_xirq_hdl** is also set up to process all events sent by the backend driver.

1.2.12.7 SEE ALSO

nk_id_get

nk_pdev_alloc

nk_ptov

vlink_lookup

nk_vtop

nk_xirq_attach

nk_xirq_detach

nk_xirq_trigger

1.2.13 VFENCE2(4D)

1.2.13.1 Cross References

Related Documents
Manual Page

1.2.13.2 NAME

vfence2 — Virtual DMA fence v2

1.2.13.3 SYNOPSIS

The vfence2 (virtual DMA fence v2) driver enables kernel and user space clients to share DMA fences between virtual machines.

vfence2 is a re-implementation of the legacy vfence driver, retroactively called v1, and offers more transparent and symmetrical operations.

1.2.13.4 FEATURES

vfence2 enables clients to share selected DMA fences through an API that exposes DMA fence export and DMA fence import commands.

Locally, DMA fences are accessed exclusively though sync_file.h handles.

The same API enables to transparently share DMA fences between processes that are either hosted by the same guest operating system (*loopback* mode) or by different guests executing in separate VMs.

vfence2 uses global fence identifiers (GIDs) to identify a fence that is exported to other processes. vfence2 GIDs are exported for a specific peer, or imported from a specific peer. Therefore, at a given time, the GID, the exporting VM id and the targeted VM id uniquely identify a fence export.

Exporting and importing is reserved to root processes.

User space operations are performed through ioctl(2)s over the */dev/vfence2* character device file. Refer to the **vfence2_uapi.h** header file for a description of vfence2 user space API and to **vfence2_kapi.h** header file for a description of vfence2 kernel space API.

Each VM involved in DMA fence sharing must enable the vfence2 driver in its kernel configuration file, by setting *CONFIG_VLX_VFENCE2=y* or *CONFIG_VLX_VFENCE2=m*

Device Drivers -> VLX virtual device support -> virtual DMA fences v2

The DMA fence sharing feature relies on a symmetric model involving vfence2 peer driver entities. Each peer can act as:

- backend, exporting DMA fences which have been locally created.
- frontend, importing DMA fences which have been allocated in peer.

There are specific optimisations which avoid inter-VM communications when a DMA fence is exported and imported back into current VM.

Peer entities collaborate through shared memory and cross-interrupts, relying on the vlink2 driver for vlink-type synchronizations. Collaboration is configured by creating a couple of peer "vfence2" compatible vlink nodes in the hypervisor device tree.

There are no user-tunable parameters.

A maximum of 1024 fences can be exported from a given guest in total.

Operations of the vfence2 driver can be observed through:

- the /proc/nk/vfence2 file
- the /proc/nk/vlink2.vfence2-fe and /proc/nk/vlink2.vfence2-be files
- kernel traces

1.2.13.4.1 vlink node

vfence2 driver parameters are configured through a "vfence2" compatible vlink node in the hypervisor device tree.

The example below implements connection between VM 2 and VM 3:

```
&vm2_vdevs {
    vfence2_23: vfence2@23 {
        compatible = "vfence2";
    };
};
&vm3_vdevs {
    vfence2@32 {
        peer-phandle = <&vfence2_23>;
    };
};
```

The example below implements a loopback connection inside VM 2:

```
&vm2_vdevs {
    vfence2_22_0: vfence2@22_0 {
        compatible = "vfence2";
        server;
    };
    vfence2@22_1 {
        peer-phandle = <&vfence2_22_0>;
        client;
    };
};
```

Note

A uni-directional vlink is enough for loopback mode, though it is also possible to use a bidirectional vlink, of which one half will remain unused.

1.2.13.5 PARAMETERS

vfence2 is implemented as a platform driver, buildable either as a built-in kernel driver or as a dynamically loadable/unloadable module.

The driver accepts no parameters.

1.2.13.6 SEE ALSO

- DMABUF buffer sharing framework

1.2.14 VG2D(4D)

1.2.14.1 Cross References

Related Documents
Manual Page

1.2.14.2 NAME

vg2d — Virtual Graphics 2D

1.2.14.3 SYNOPSIS

The virtual g2d (vg2d) back-end and front-end drivers, when paired together, allow userspace clients to access the functionality of Graphics 2D from a virtual machine which does not host the physical hardware. This man page describes both the frontend and the backend virtual Graphics 2D drivers. The vg2d backend driver receives requests from the frontend vg2d driver and forwards them to the underlying native G2D via standard Linux driver API.

1.2.14.4 FEATURES

The vg2d frontend driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual G2D frontend interface

The vg2d backend driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual G2D backend interface

Both drivers can be compiled as loadable modules or as embedded drivers.

Virtual G2D backend and frontend communicate using VRPC(4D). One VRPC link must be declared in the platform device tree for each (backend, frontend) pair. The example below declares two vrpc links (VM2, VM3) using the virtual link framework (see VRPC(4D) manual page for more details).

```
&vm2_vdevs {
    vg2d_be: vg2d-be {
        compatible = "vrpc";
        server;
        info = "vg2d,16384";
    };
    vfimg2d_be: vfimg2d-be {
        compatible = "vrpc";
        server;
        info = "vfimg2d,16384";
    };
};
```

```
&vm3_vdevs {
    vg2d-fe {
        peer-phandle = <&vg2d_be>;
        client;
        info = "vg2d";
    };
    vfimg2d-fe {
        peer-phandle = <&vfimg2d_be>;
        client;
        info = "vfimg2d";
    };
};
```

The vg2d backend driver optionally supports running in Google's Generic Kernel Image mode, where it is loaded dynamically and only uses a Google-approved subset of exported Linux kernel symbols. For this, it must be compiled with the `CONFIG_VLX_VG2D_BE_VGKI` configuration option. It then uses the VGKI kernel-mode API instead of non-GKI calls, requires the presence of the VGKI(4D) driver to load and of the vgki-helper(8) process to perform requests. The vg2d frontend driver is GKI compatible by default.

1.2.14.5 PARAMETERS

The *info* property can be used on the backend side to change default configuration. The syntax is: `vlink_name[,pmem_size]`

1.2.14.6 SEE ALSO

VRPC(4D)

VION(4D)

VFENCE2(4D)

VGKI(4D) - a driver internally used by vg2d backend for Generic Kernel Image compatibility

1.2.15 VGKI(4D)

1.2.15.1 Cross References

Related Documents
Manual Page

1.2.15.2 NAME

vgki driver — Virtual Generic Kernel Image driver

1.2.15.3 SYNOPSIS

The `vgki` driver implements a kernel mode API (KAPI) which replaces calls not allowed by the restricted Android GKI API.

1.2.15.4 FEATURES

The following KAPI calls are offered, either replacing the indicated kernel calls and taking the same arguments, or having POSIX-like arguments:

VGKI call	non-GKI call
`vgki_kapi_filp_open()`	Kernel `filp_open()`
`vgki_kapi_kernel_thread()`	Kernel `kernel_thread()`
`vgki_kapi_open()`	POSIX `open(2)`
`vgki_kapi_close()`	POSIX `close(2)`
`vgki_kapi_mmap()`	Simplified POSIX `mmap(2)`
`vgki_kapi_munmap()`	POSIX `munmap(2)`

Calls are actually performed by the vgki-helper(8) process, which communicates with the `vgki` driver using a private API.

1.2.15.4.1 CONFIGURATION

The `vgki` driver does not need configuration. However, it requires the presence of the vgki-helper(8) process, which must be loaded for `vgki` driver calls to complete.

1.2.15.5 PARAMETERS

The `vgki` driver does not have parameters.

1.2.15.6 SEE ALSO

- `Generic Kernel Image | Android Open Source Project`

1.2.16 VI2C(4D)

1.2.16.1 Cross References

Related Documents
Manual Page

1.2.16.2 NAME

vI2C — Virtual Inter-Integrated Circuit (I2C)

1.2.16.3 SYNOPSIS

The I2C virtualization consists of a pair of drivers running inside QNX and Linux guests. When these drivers have been paired together it allows some I2C slaves to be accesses from IVI domain.

1.2.16.4 FEATURES

As it was previously mentioned the frontend driver, running in Linux guests, is implemented as a device driver and so it introduces its own Kconfig. To build the vI2C frontend **VLX_VI2C_FE** must be enabled.

Device Drivers -> VLX virtual device support -> VLX_VI2C_FE

The backend driver, running in a QNX guest, is implemented as an application which could be started from the command line and so it expects its configuration as command line arguments. For more information regarding how to start and setup the backend driver Virtual I2C slave access policy.

1.2.16.4.1 Virtual I2C adapter declaration

Under Linux the I2C buses, adapters are typically represented by nodes in the device tree. Example of such a device tree configuration is given below.

```
/{
    vi2c5: vi2c@5 {
        compatible = "vl,i2c-adapter";
        /*
         * This string must be the same as the
         * compatible string inside the vlink node.
         */
        vl,vlink = "vi2c5";
        /*
         * I2C slaves should be declared below.
         */
    };

    /*
     * Because of this alias the I2C core in
     * Linux will register /dev/i2c-5.
     */
    aliases {
        i2c5 = &vi2c5;
    };
};
```

The virtual I2C frontend driver relies on some mandatory properties in the device tree. For each vI2C bus a compatible property must be set to **"vl,i2c-adapter"**. The other important property in the device tree is **"vl,vlink"**. This property must match the name of the vlink between each vI2C backend - frontend peers.

1.2.16.4.2 Virtual I2C slave declaration

As described in `Instantiating I2C devices` there are many ways to attach slave devices to a particular vI2C bus.

Below is given one of the ways that the vI2C frontend driver supports to bind a slave device to a particular vI2C bus. This is done by a child node declaration inside the vI2C adapter's node.

```
/{
    vi2c1: vi2c@1 {
        compatible = "vl,i2c-adapter";
        /*
         * This string must be the same as the
         * compatible string inside the vlink node.
         */
        vl,vlink = "vi2c1";
        /*
         * Flash memory slave device on I2C bus 1.
         */
        flash@50 {
            compatible = "atmel,24c256";
            reg = <0x50>;
        };
    };

    aliases {
        i2c1 = &vi2c1;
    };
};
```

1.2.16.4.3 Virtual I2C vlink declaration

By design each virtualized I2C bus, adapter requires a separate vlink declaration. Below is given such a vlink declaration for a pair of backend - frontend drivers, dedicated to serve the requests for I2C adapter 5.

Note

In order to virtualize multiple I2C adapters separate instances of the backend driver must be ran, one for each adapter.

```
vm2_vdevs {
    vi2c0_be: vi2c0@be {            // vi2c back-end
        compatible = "vi2c5";       // vi2c front-end for I2C bus 5
    };
};
vm3_vdevs {
    vi2c0@fe {              // vi2c front-end
        peer-phandle = <&vi2c0_be>; // peer vlink
    };
};
```

Note

The name of the vlink must be the same as the string in **"vl,vlink"** property from the vI2C bus declaration.

1.2.16.4.4 Virtual I2C slave access policy

The vI2C frontend driver forwards the requests from/to the client drivers in IVI domain to/from the vI2C backend driver. When a request has been received, the backend driver must decide whether to forward the request to the native driver or to send back a response with status field set to permission error.

By design the backend driver accepts the following command line arguments:

- I2C bus, adapter identifier: **-d**$<$**N**$>$, where $<$**N**$>$ stands for a bus id.
- vlink name: **-l**$<$**vlink**$>$, where $<$**vlink**$>$ is the name of the vlink the backend driver will search for.
- Slave access policy: **-a**$<$**vm1=**$*|$**addr1,...,addrN;...;vmN=**$*|$**addr1,...,addrN**$>$

In the next few lines are given multiple examples on how to run the vI2C backend driver with a proper slave access policy.

- Starting the backend driver to allow access from VM3 to slaves 0x39 and 0x3c on I2C bus 1.

  `devc-vi2c-be -d1 -l"vi2c1" -a"vm3=0x39,0x3c"`

- Starting the backend driver to allow access to all slaves from VM3 on I2C bus 3.

  `devc-vi2c-be -d3 -l"vi2c3" -a"vm3=*"`

- Starting the backend driver to allow accesses from VM3 to slave 0x10 and from VM4 to slave 0x6f on I2C bus 5.

  `devc-vi2c-be -d5 -l"vi2c5" -a"vm3=0x10;vm4=0x6f"`

- Starting the backend driver to deny access to all slaves from all VMs.

  `devc-vi2c-be -d3 -l"vi2c3"`

Note

Without an access policy the vI2c be will deny access to all slaves for a particular adapter.
By requirement the vI2C backend driver won't allow access to the same slave from multiple guests.

1.2.16.5 SEE ALSO

Virtual Message Queue Device driver interface layer for virtual drivers**(4D)**

Linux kernel I2C and SMbus subsystem

Generic device tree bindings for I2C buses

QNX I2C framework

1.2.17 VION(4D)

1.2.17.1 Cross References

Related Documents
Manual Page

1.2.17.2 NAME

vion — Virtual ION

1.2.17.3 SYNOPSIS

The virtual ION (vion) driver enables userspace clients to share memory buffers allocated by the ION memory allocator.

1.2.17.4 FEATURES

The ION memory allocator enables userspace clients to allocate DMABUF buffers that use dedicated memory heaps as their backing storage.

vION enables clients to share these buffers through a secured userspace API that exposes buffer export and buffer import commands. The same API enables to transparently share ION buffers between processes that are either hosted by the same guest operating system or by different guests executing in separate domains.

vION uses global buffer identifiers to identify the ION buffers that are exported to other processes. vION's system-wide GIDs uniquely identify ION buffers regardless of the domain it was allocated in.

The API exposed by the ION driver can be secured by the optional use of buffer credentials. Credentials are opaque data structures that match global buffer identifiers. Buffer credentials consists in an 16-byte UUID set to a random value at the buffer's export time. When vION is set to operate in secure mode, vION clients are required to present these credentials in addition to global identifiers to import ION buffers.

Refer to the **vion_uapi.h** header file for a description of vION's userspace API.

Each domain involved in buffer sharing MUST enable the vION driver in its kernel configuration file:

Device Drivers -> VLX virtual device support -> virtual ION

The buffer sharing feature relies on an asymmetric model involving a couple of vION peer entities:

- the vION back-end entity exports buffers that have been locally allocated,
- the vION front-end entity imports buffers that have been allocated on the back-end side.

Peer entities collaborate together through a vRPC channel whose configuration is controlled by a couple of peer "vrpc" compatible vlink nodes of the hypervisor device tree.

1.2.17.4.1 vRPC node

vION driver parameters that are related to the underneath vRPC channel are configured through a "vrpc" compatible node in the virtual platform device tree.

The example below implements an asymmetric scheme where domain#3 can import buffers exported by domain#2:

```
&vm2_vdevs {
        vion_be23: vion-be {
                compatible = "vrpc";
                server;
                info =  "vion,130152";
                critical;
        };
};
&vm3_vdevs {
        vion-fe23 {
                peer-phandle = <&vion_be23>;
                client;
                info =  "vion";
        };
};
```

The "vrpc" compatible node has the following properties:

- the *compatible* property is set to "vrpc" and identifies a vRPC node,
- the *client/server* property defines the vION channel back-end/front-end orientation, the server domain acting as the back-end, the client domain acting as the front-end,
- the *critical* property which allows vION resilience to back-end restart,
- the *info* property identifies the client driver ("vion" in this case). On the server side it defines the vION channel pmem size (i.e. the maximal size of the messages that can be exchanged by vION peers), the syntax is: "info = vlink_name[,pmem_size]",

Note

vION peer entities are not involved in buffer content exchange, hence the channel pmem size has nothing to do with buffer size. Instead, vION peer entities exchange buffer layouts (i.e. arrays of (paddr, size) for all the physically contiguous chunks that constitute a buffer). The buffer layout size is not completely dictated by the buffer size as a big buffer can be made-up of a single chunk, and a small buffer can be made-up of many chunks, but since a buffer cannot be made-up of chunks smaller than a page, there exists a low limit of the layout size, given the buffer size. vION enables clients to import several buffers layout in a single transaction, this facility is only limited by the pmem size. Consequently, pmem size must be adjusted appropriately to maximize performances: minimal pmem size must allow to convey a single buffer at least, maximal pmem size must allow to convey, if not all, most of the imports in a single transaction. Given the maximal buffer size to be shared, and the maximal number of buffers to be imported in a single transaction, the required pmem size can be calculated by the formula: pmem-size-in-bytes = 16 + (112 + buffer-size-in-bytes / 4096 $*$ 16) $*$ number-of-buffers (e.g. for a maximal buffer size of 32 MB, and a number of buffers of 5, the channel pmem size must be higher or equal to: 16 + (112 + 32 $*$ 1024^2 / 4096 $*$ 16) $*$ 5 = 655936 Bytes).

Two asymmetric channels can be combined together to achieve a symmetric buffer sharing scheme between two domains.

The example below implements a symmetric scheme where domain#2 and domain#3 can both export and import buffers from each other domain:

```
&vm2_vdevs {
        vion_be23: vion-be {
                compatible = "vrpc";
                server;
                info =  "vion,130152";
                critical;
        };
        vion-fe32 {
                peer-phandle = <&vion_be32>;
                client;
                info =  "vion";
        };
};
&vm3_vdevs {
        vion_be32: vion-be {
                compatible = "vrpc";
                server;
                info =  "vion,130152";
                critical;
        };
        vion-fe23 {
                peer-phandle = <&vion_be23>;
                client;
                info =  "vion";
        };
};
```

In order to import buffers from an exporter domain, the importer domain MUST include the exporter domain memory as a reserved memory area in its device tree. The example below illustrates how the domain#3 device tree has to be modified to implement the asymmetric buffer sharing scheme mentioned above:

```
reserved-memory {
        #address-cells = <2>;
        #size-cells = <1>;
        ranges;
        vm-memory@2 {
                compatible = "vl,vm-memory";
                reg = <0 0x0 0x0>;
                vl,vm-id = <2>;
        };
};
```

Additionally, the domain#2 device tree must be modified as follows to implement the symmetric buffer sharing scheme mentioned above:

```
reserved-memory {
        #address-cells = <2>;
        #size-cells = <1>;
        ranges;
        vm-memory@3 {
                compatible = "vl,vm-memory";
                reg = <0 0x0 0x0>;
                vl,vm-id = <3>;
        };
};
```

1.2.17.4.2 vION node

vION driver parameters that are not related to the underneath vRPC channel are configured through a "vl,vion" compatible node in the guest device tree.

```
vion {
        compatible = "vl,vion";
        memory-granting-flags = "read","write","cpu","dma","nolock";
};
```

The "vl,vion" compatible node has the following properties:

- the *compatible* property is set to "vl,vion" and identifies a vION node,
- the *memory-granting-flags* property defines the access permission granted to any vION buffer imported by this domain, regardless the exporting domain. It is a multi-string property which accepts any combination of the following elementary strings. When an elementary string is not present in the property, the associated access permission is denied.
  - "read": authorizes read access,
  - "write": authorizes write access,
  - "dma": authorizes DMA access,
  - "cpu": authorizes CPUs access.
  - "nolock": enables the 'nolock' optimization in the hyp_call_vm_mem_grant() hypervisor call.

Note

Both "vl,vion" node and *memory-granting-flags* property are optional. when either the "vl,vion" node or the *memory-granting-flags* property is absent, the default value memory-granting-flags = "read","write","cpu","dma"; is used.

The example below illustrates the case where vION buffers are accessible for read and write by the DMA only with the 'nolock' optimization enabled.

```
vion {
        compatible = "vl,vion";
        memory-granting-flags = "read","write","dma","nolock";
};
```

1.2.17.5 PARAMETERS

vION is implemented as a platform driver buildable either as a built-in kernel driver or as dynamically loadable/unloadable module.
The driver accepts the following parameters:

- **vion_secure**: when this parameter is set a non-zero-value, vION operates in secure mode. In that mode, vION clients are required to present credentials in addition to global identifiers to import ION buffers.
  Conversely, setting this parameter to zero switches vION to unsecure mode. In that mode, only the global identifiers are necessary to import ION buffers.
- **vion_debug**: this parameter controls the verbosity level of the driver.
  Setting this parameter to 0 disables all the traces, setting it to 1 enables only error messages, setting it to 2 enables additional informational messages, and setting it to 3 enables all the debug traces.

1.2.17.6 SEE ALSO

- DMABUF buffer sharing framework
- vrpc

1.2.18 VLINK_LIB(4D)

1.2.18.1 NAME

vlink_lib — Primitives provided by the vlink wrapper library

1.2.18.2 SYNOPSIS

#include <vlx/vlink-lib.h>

This library provides a set a primitives for virtual link management to virtual drivers. This library is used by other libraries and virtual drivers.

1.2.18.3 FEATURES

The virtual link wrapper library is not a driver and is linked with all virtual drivers.

1.2.18.4 VIRTUAL LINK WRAPPER LIBRARY OBJECTS

The virtual link wrapper library interface is given in the **vlink-lib.h** file.

1.2.18.4.1 VIRTUAL LINK DRIVER DESCRIPTOR

The first object handled by this library is a virtual link driver descriptor defined as follows:

```
typedef int  (*VlinkDrvInit)    (struct VlinkDrv*);
typedef void (*VlinkDrvCleanup) (struct VlinkDrv*);
typedef int  (*VlinkInit)       (struct Vlink*);

typedef struct VlinkDrv {
    /*
     * Public driver information - provided by the driver.
     */
    const char*       name;
    VlinkDrvInit      init;
    VlinkDrvCleanup   cleanup;
    VlinkInit         vlink_init;
    unsigned int      flags;
    /*
     * Public driver information &mdash; provided by the vlink library.
     */
    unsigned int      nr_units;
    /*
     * Private driver information &mdash; provided by the driver.
     */
    void*             private;
    /*
     * Internal vlink library data.
     */
    unsigned int      nr_clients;
    unsigned int      nr_servers;
    unsigned int      state;
    NkXIrqId          sysconf_id;
    VlListHead        vlinks;
} VlinkDrv;
```

The first public **name** field contains the name of the virtual driver.

The second public **init** field holds the virtual driver initialization callback routine (provided by the virtual driver) and invoked by the **vlink_drv_startup(VlinkDrv∗ drv)** virtual link library primitive (see section VIRTUAL LINK DRIVER STARTUP for further details).

The third public **cleanup** field holds the virtual driver cleanup callback routine (provided by the virtual driver) and invoked by the **vlink_drv_cleanup(VlinkDrv∗ drv)** virtual link library primitive (see section VIRTUAL LINK DRIVER CLEANUP for further details), or by the virtual driver itself.

The fourth public **vlink_init** field holds the virtual link initialization callback routine (provided by the virtual driver) invoked by the **vlink_drv_startup(VlinkDrv∗ drv)** virtual link library primitive.

The fifth public **flags** field holds the type of the virtual driver:

VLINK_DRV_TYPE_CLIENT

Client virtual link driver

VLINK_DRV_TYPE_SERVER

Server virtual link driver

VLINK_DRV_TYPE_SYMMETRIC

Both server and client virtual link driver

The sixth public **nr_units** field holds the numbers of servers held in **nr_servers** field (see below) if the virtual link driver is of server type (according to the content of the **flags** field), or the number of clients held in **nr_clients** field (see below) if the virtual link is of client type. Note that if the virtual link type is equal to *VLINK_DRV_TYPE_SY*↩ *MMETRIC*, this field is set to the number of servers.

The seventh **private** field is used to store private data of the virtual driver. This field is opaque for the virtual link library routines. This field is used by virtual driver to retrieve their private data when a callback is invoked by the virtual link library.

The next couple of internal fields **nr_clients** and **nr_servers** are respectively used to store the number of clients and the number of servers. These fields cannot be changed by the virtual link driver.

The next internal **state** field holds the current status of the virtual link driver:

VLINK_DRV_CLEAN

This state is set when the **vlink_drv_cleanup(VlinkDrv∗ drv)** has successfully performed the requested operation (see section VIRTUAL LINK DRIVER CLEANUP for further details)

VLINK_DRV_STOPPED

This state is set when the **vlink_drv_shutdown(VlinkDrv∗ drv)** has ended its processing (see section VIRTUAL LINK DRIVER SHUTDOWN for further details)

VLINK_DRV_STARTED

This state is set when the **vlink_drv_startup(VlinkDrv∗ drv)** has ended its processing (see section VIRTUAL LINK DRIVER STARTUP for further details)

VLINK_DRV_PROBED

This state is set when the **vlink_drv_probe(VlinkDrv∗ drv)** has successfully ended its processing (see section VIRTUAL LINK DRIVER PROBE for further details)

The next internal **sysconf_id** field holds the cross interrupt number corresponding to the *NK_XIRQ_SYSCONF* event through the invocation of the NKDDI **nk_xirq_attach(3D)** primitive.

Finally, the last **vlinks** field holds a list of *Vlink* object explained below in this section.

1.2.18.4.2 VIRTUAL LINK DESCRIPTOR

The next object exported by the virtual link library is a virtual link descriptor: *Vlink* whose layout is described below:

```
typedef struct Vlink {
    /*
     * Public device information - provided by the vlink library.
     */
    VlinkDrv*              drv;
    NkDevVlink*            nk_vlink;
    struct Vlink*          sym_vlink;
    unsigned int           server;
    unsigned int           unit;
    NkOsId                 id;
    NkOsId                 id_peer;
    /*
     * Private device information - provided by the driver.
     */
    void*                  private;
    /*
     * Internal vlink library data.
     */
    volatile int*          nk_state;
    volatile int*          nk_state_peer;
    volatile unsigned int  state;
    volatile unsigned int  state_target;
    VlSpinlock             state_lock;
    VlAtomic               users;
    VlThread*              admin_thread;
    VlAtomic               admin_event;
    VlinkWaitQueue         admin_event_wait;
    VlinkWaitQueue         admin_comp_wait;
    VlinkSem               sessions_lock;
    VlinkSem               sessions_start_lock;
    unsigned int           sessions_started;
    VlinkWaitQueue         sessions_end_wait;
    VlAtomic               sessions_count;
    VlListHead             xirqs;
    VlListHead             sessions;
    VlListHead             ops[VLINK_OP_NR];
    VlListHead             link;
} Vlink;
```

The first public **drv** field holds a pointer to a *VlinkDrv* object which is described in the section VIRTUAL LINK DRIVER DESCRIPTOR.

The second public **nk_vlink** field is a pointer to a *NkDevVlink* object whose fields are explained in the NKDDI vlink_lookup primitive.

The third public **sym_vlink** field is a pointer to a *Vlink* object. This field is set to a null pointer if the virtual link driver type is server or client. If the virtual link driver is for both server and client (*VLINK_DRV_TYPE_SYMMETRIC*, see section VIRTUAL LINK DRIVER DESCRIPTOR for further details), this field set to the client virtual link if the current *Vlink* is a server and conversely. Both server and client must have the same **id** and **id_peer** field values (see below for further details). In other words, the client and the server must run on the same VM.

This fourth public **server** field is set to **1** if the virtual link driver is a server type (*VLINK_DRV_TYPE_SERVER*), otherwise this field is set to **0**.

The fifth public **unit** field is set to respectively **drv->nr_servers** for a server type, or to **drv->nr_clients** for a client type (see section VIRTUAL LINK DRIVER DESCRIPTOR for further details).

The couple of public fields **id** and **id_peer** are respectively set to server VM identifier (that is, the VM where the server is running), and to the client VM identifier (that is, the guest OS where the client is running).

All these fields are updated by the **vlink_drv_probe(VlinkDrv* drv)** primitive through **vlink_create()** and **vlink_↩ attach_sym_vlinks()** internal functions of the virtual link library.

The sixth **private** field is used to store private data of the virtual link driver. This field is opaque for the virtual link library routines. This field is used by virtual driver to retrieve their private data when a callback is invoked by the virtual link library.

All other fields of this object are private to the virtual link library recording the current virtual link states, thread for processing events, various synchronization objects (semaphores, spin lock), session counters, and so on.

1.2.18.4.3 VIRTUAL LINK SESSION DESCRIPTOR

A virtual link object may have multiple sessions. The virtual link session descriptor and more precisely a *VlinkSession* data structure is described below:

```
typedef int (*VlinkSessionOp) (struct VlinkSession* session);

typedef struct VlinkSession {
    Vlink*                  vlink;
    volatile unsigned int   state;
    VlAtomic                entered;
    VlAtomic                refcount;
    void*                   private;
    VlinkSessionOp          op_abort;
    VlListHead              link;
} VlinkSession;
```

The first **vlink** field is a valid pointer to a *Vlink* object whose fields are described in section VIRTUAL LINK DESC↩
RIPTOR.

The second **state** field is the current state of the virtual link session:

VLINK_SESSION_NEW

The virtual link session is successfully created (see section VIRTUAL LINK SESSION CREATE for further details)

VLINK_SESSION_ALIVE

The virtual link session is set alive when the corresponding virtual link is up (*VLINK_UP* state, see section VIRTUAL LINK SESSION CREATE for further details)

VLINK_SESSION_ABORTED

The virtual link session is set to this value through the internal function **vlink_abort()** when the virtual link is no longer on (that is, the peer state virtual link is in *NK_DEV_VLINK_OFF* state)

The next couple of fields **entered** and **refcount** are respectively modified when entering (atomically incremented) or leaving a session (atomically decremented), and when a get operation (atomically incremented) or a put operation (atomically decremented) are performed on a virtual link session.

The **private** field holds private data belonging to the virtual link driver. This field is opaque for the virtual link library.

The **op_abort** field is used to record the abort routine provided by the virtual link driver when a virtual link is aborted (*VLINK_SESSION_ABORTED*).

The last **link** field is used to record a list of *Vlink* objects in order to retrieve the associated sessions (**session** field of *Vlink* object).

All these fields are updated when a session is created with the **vlink_session_create()** primitive (see section VIRTUAL LINK SESSION CREATE for further details).

The remaining sections are devoted to virtual link driver primitives, virtual link sessions primitives and macros provided by this library.

1.2.18.5 VIRTUAL LINK DRIVER PROBE PRIMITIVE

1.2.18.5.1 SYNOPSIS

This primitive is used to probe the virtual driver name and virtual link.

#include <vlx/vlink-lib.h>

int **vlink_drv_probe** (VlinkDrv∗ *drv*);

1.2.18.5.2 DESCRIPTION

This primitive is invoked at module initialization when a virtual driver is starting up.

1.2.18.5.3 PARAMETERS

The single **drv** argument is a valid pointer to a *VlinkDrv* object whose some minimal information have been previously updated by the virtual driver before invoking this virtual link primitive. The fields to update are listed below:

name

The name of the virtual driver (a string of characters)

init

The entry point of the virtual driver initialization callback routine (see section VIRTUAL LINK DRIVER STARTUP for further details)

cleanup

The entry point of the virtual driver cleanup callback routine (see section VIRTUAL LINK DRIVER CLEANUP for further details)

vlink_init

The entry point of the virtual link initialization callback routine (see section VIRTUAL LINK DRIVER STARTUP for further details)

flags

The type of the virtual driver (server, client or both)

Note that these fields are mandatory to update otherwise fatal error or unpredictable behavior may occur.

This primitive is based on the NKDDI nk_vlink_lookup primitive to retrieve the virtual device driver tree, and the corresponding virtual links. After a successful probe operation the current state of the virtual link driver is set to *VLINK_DRV_PROBE*.

1.2.18.5.4 RETURN VALUES

In case of success, this primitive returns 0, otherwise the following error codes are returned:

-ENOMEM

There are no more resources to perform the requested operation

-EINVAL

For symmetric virtual links, either the client or the server virtual link cannot be retrieved in the virtual device tree

1.2.18.6 VIRTUAL LINK DRIVER STARTUP PRIMITIVE

1.2.18.6.1 SYNOPSIS

This primitive starts up a virtual driver.

#include <vlx/vlink-lib.h>

int **vlink_drv_startup** (VlinkDrv∗ *drv*);

1.2.18.6.2 DESCRIPTION

This primitive starts up a virtual link driver after successfully calling the **vlink_drv_probe(VlinkDrv∗ drv)** primitive. This primitive invokes the initialization routine with the following prototype:

```
typedef int(*VlinkDrvInit)(struct VlinkDrv* drv);
```

This routine has been provided by the virtual link driver when **vlink_drv_probe(VlinkDrv∗ drv)** has been invoked. It is invoked by the virtual link library with a valid pointer to a *VlinkDrv* object (same pointer value than the **vlink_drv_startup(VlinkDrv∗ drv)** primitive). It should allocate the necessary resources to start up the virtual link driver and returns 0 in case of success. Otherwise a non null error code is returned, and the **vlink_drv_startup(VlinkDrv∗ drv)** aborts immediately its processing.

For each virtual link, this primitive invokes the virtual link initialization routine also provided by the virtual link driver upon **vlink_drv_probe(VlinkDrv∗ drv)** invocation. This virtual link initialization routine has the following prototype:

```
typedef int(*VlinkInit)(struct Vlink* vlink);
```

This callback routine is invoked with a valid pointer to a *Vlink* object. It should initialize the requested virtual link. In case of success, it returns 0, otherwise a non null error code is returned.

This primitive creates a thread to process events for each virtual link, all virtual link sessions are started, and a cross interrupt for *NK_XIRQ_SYSCONF* is attached.

If all virtual links have been correctly initialized and the corresponding thread has been successfully created, their states are set to *VLINK_STOPPED*, otherwise their states are set to *VLINK_CLEAN* (**state** field of a *Vlink* object).

In case of success, the current state of the virtual link driver is set to *VLINK_DRV_STARTED*. This state is also set if some virtual link have not been correctly initialized (not in the *VLINK_STOPPED* state).

1.2.18.6.3 PARAMETERS

This routine is invoked with a valid pointer to a *VlinkDrv* object. This object has been initialized with the minimum information described in section VIRTUAL LINK DRIVER PROBE PRIMITIVE and after calling the **vlink_drv_probe(VlinkDrv∗ drv)** primitive.

1.2.18.6.4 RETURN VALUES

In case of success, this primitive returns 0, otherwise the following error codes are returned

-ENOMEM

There are no more resources to perform the requested operation

all non null error codes

These non null error codes are returned by the virtual link driver initialization routine

1.2.18.7 VIRTUAL LINK DRIVER SHUTDOWN PRIMITIVE

1.2.18.7.1 SYNOPSIS

This primitive shuts down the virtual link driver and all related virtual links.

#include <vlx/vlink-lib.h>

int **vlink_drv_shutdown** (VlinkDrv∗ *drv*);

1.2.18.7.2 DESCRIPTION

This primitive is invoked by a virtual link driver when the module is going to be removed. All virtual links are set to the *VLINK_STOPPED* (**state_target** field of *Vlink* object) and all related events (cross interrupts) have been sent to signal to client/server virtual link driver that the current one is shutting down.

After this call, the current state of the virtual link driver is set to *VLINK_DRV_STOPPED*.

1.2.18.7.3 PARAMETERS

This primitive is invoked with a valid pointer to a *VlinkDrv* object.

1.2.18.7.4 RETURN VALUES

This primitive always returns 0.

1.2.18.8 VIRTUAL LINK DRIVER CLEANUP PRIMITIVE

1.2.18.8.1 SYNOPSIS

This primitive is invoked to release all resources allocated for a virtual link driver.

#include <vlx/vlink-lib.h>

void **vlink_drv_cleanup** (VlinkDrv∗ *drv*);

1.2.18.8.2 DESCRIPTION

This primitive is invoked last after **vlink_drv_shutdown(VlinkDrv∗ drv)** in order to release all resources allocated by the virtual link library for a virtual link driver. The cross interrupt for *NK_XIRQ_SYSCONF* event is released (based on the nk_xirq_detach primitive), all virtual links are set in the *VLINK_CLEAN* status and the corresponding thread for managing event is deleted.

In addition, the virtual link driver cleanup routine provided when **vlink_drv_probe(VlinkDrv∗ drv)** has been invoked, is called by this primitive. The prototype of this callback routine is:

```
typedef void (*VlinkDrvCleanup)(struct VlinkDrv* drv);
```

This routine must deallocate all private resources of the virtual link driver. No relevant value is returned by this routine.

Finally the current state of the virtual link driver is set to *VLINK_DRV_CLEAN*.

1.2.18.8.3 PARAMETERS

This primitive is invoked with a valid pointer to a *VlinkDrv* object.

1.2.18.8.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.18.9 VIRTUAL LINK SESSION CREATE PRIMITIVE

1.2.18.9.1 SYNOPSIS

This primitive creates a session.

#include <vlx/vlink-lib.h>

int **vlink_session_create** (Vlink∗ *vlink*, void∗ *private*, VlinkSessionOp *op_abort*, VlinkSession∗∗ *psession*);

1.2.18.9.2 DESCRIPTION

This primitive is invoked to create a session allowing to perform operations on a virtual link.

1.2.18.9.3 PARAMETERS

The first *vlink* argument is a valid pointer to a *Vlink* object. This object is provided by the virtual link library primitive when the callback routine for initializing virtual links are invoked by **vlink_drv_startup(VlinkDrv∗ drv)** primitive (see section VIRTUAL LINK DRIVER STARTUP for further details).

The second *private* argument is provided by the caller in order to store private data in the returned *VlinkSession* pointer in case of success. This field may be a null pointer if no private data need to be recorded.

The third *op_abort* argument is the entry point of the abort session callback routine provided by the virtual link driver. This callback has the following prototype:

```
typedef int (*VlinkSessionOp) (struct VlinkSession* session);
```

This handler is invoked by the virtual link library if the associated virtual link is turned off (*NK_DEV_VLINK_OFF* the client/server is shutdown). This handler is invoked with a valid pointer to a session descriptor (*VlinkSession* object). This handler should return 0 is case of success, otherwise a non null error code should be returned.

This argument can be set to a null pointer if the virtual link driver has no specific operation to execute when a virtual link is down.

The last *psession* argument is the address of a valid pointer which is updated in case of success with a newly created and initialized *VlinkSession* object, and the current virtual link session state is set to *VLINK_SESSION_NEW* (**state** field of the *VlinkSession* object).

1.2.18.9.4 RETURN VALUES

In case of success, this primitive returns 0. In case of error, the following error codes are returned:

-ERESTARTSYS

The same operation is in progress on this virtual link

-ENXIO

The virtual link is in the *VLINK_CLEAN* or if **vlink_drv_shutdown(VlinkDrv∗ drv)** has been invoked

-ENOMEM

If there is not enough memory to allocate a *VlinkSession* object

1.2.18.10 VIRTUAL LINK SESSION RELEASE PRIMITIVE

1.2.18.10.1 SYNOPSIS

This primitive releases a previously created session.

#include <vlx/vlink-lib.h>

int **vlink_session_release** (VlinkSession∗ *session*);

1.2.18.10.2 DESCRIPTION

This primitive is invoked to release a session previously created with **vlink_session_create()** primitive. The session must be not be in used by the virtual link driver (*refcount* field of *VlinkSession* object must be equal to 0.

1.2.18.10.3 PARAMETERS

The single *session* argument is a valid pointer to a *VlinkSession* object previously returned upon creation.

1.2.18.10.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.18.11 VIRTUAL LINK REGISTER ONE OPERATION PRIMITIVE

1.2.18.11.1 SYNOPSIS

This primitive records a callback routine provided by the virtual link driver.

#include <vlx/vlink-lib.h>

int **vlink_op_register** (Vlink∗ *vlink*, unsigned int *idx*, VlinkOp *op*, void∗ *cookie*);

1.2.18.11.2 DESCRIPTION

This primitive records a callback routine provided by the virtual link driver in order to perform one of the following operation on a virtual link:

VLINK_OP_RESET (0)

Reset a virtual link

VLINK_OP_START (1)

Start a virtual link

VLINK_OP_ABORT (2)

Abort a virtual link

VLINK_OP_STOP (3)

Stop a virtual link

VLINK_OP_CLEANUP (4)

Cleanup a virtual link

VLINK_OP_UP (5)

Set a virtual link up

This callback routine is invoked by the **vlink_op_perform()** virtual link library (see section VIRTUAL LINK OPERATION PERFORM for further details).

1.2.18.11.3 PARAMETERS

The first *vlink* argument is a valid pointer to a *Vlink* object.

The second *idx* argument is the index of the callback routine. This index designates the type of operation (see above) and thus must belong to the range 0-5 (must be less than *VLINK_OP_NR* set to 6 per default).

The third *op* argument is a valid pointer to the callback routine whose prototype is defined below:

```
typedef int (*VlinkOp) (struct Vlink* vlink, void* cookie);
```

This callback is invoked by the virtual link library with a valid pointer to a *Vlink* object and with the *cookie* argument given either through the **vlink_op_register()** or **vlink_ops_register()** virtual link library primitives.

1.2.18.11.4 VALUES

In case of success, this primitive returns 0, otherwise *-ENOMEM* is returned if there is not enough memory to execute this service.

1.2.18.12 VIRTUAL LINK RECORD MULTIPLE OPERATIONS PRIMITIVE

1.2.18.12.1 SYNOPSIS

This primitive records array of callback routines to perform all possible operations on a virtual link.

#include <vlx/vlink-lib.h>

int **vlink_ops_register** (Vlink∗ *vlink*, VlinkOpDesc∗ *ops*, void∗ *cookie*);

1.2.18.12.2 DESCRIPTION

This primitive records an array of callback routines to perform all possible operations described in section VIRTUAL LINK REGISTER ONE OPERATION: reset, start, abort, stop, cleanup and up.

1.2.18.12.3 PARAMETERS

The first *vlink* argument is a valid pointer to a *Vlink* object.

The second *ops* argument is a pointer to an array of virtual link operation descriptor of which each record has the following layout:

```
typedef struct VlinkOpDesc {
    unsigned int     idx;
    VlinkOp          op;
} VlinkOpDesc;
```

The first **idx** index is the desired operation and belong to range 0-5 (see section VIRTUAL LINK REGISTER ONE OPERATION for further details). The second **op** field holds either a valid pointer to a callback routine (*VlinkOp* object) or a null pointer. The primitive stops when a null pointer is detected. So, this array must always be ended with an invalid slot containing a pair of null values.

The last *cookie* argument is the second argument given to the callback routine. This value may be a non-null pointer if the callback routine needs private data.

This primitive is based on **vlink_op_register()** primitive, for each callback routine this latter primitive is invoked.

1.2.18.12.4 RETURN VALUES

The return codes are the same as **vlink_op_register()** primitive.

1.2.18.13 VIRTUAL LINK OPERATION PERFORM PRIMITIVE

1.2.18.13.1 SYNOPSIS

This primitive performs a previously recorded callback operation applied to a virtual link.

#include <vlx/vlink-lib.h>

void **vlink_op_perform** (Vlink∗ *vlink*, unsigned int *idx*);

1.2.18.13.2 DESCRIPTION

This primitive performs a previously recorded callback operation through **vlink_op_register()** or **vlink_ops_register()** virtual link primitives (see sections VIRTUAL LINK REGISTER ONE OPERATION and VIRTUAL LINK RECORD MULTIPLE OPERATIONS for further details).

1.2.18.13.3 PARAMETERS

The first **vlink** argument is a valid pointer to a *Vlink* object.

The second **idx** argument is the requested virtual link operation: reset, start, abort, stop, cleanup and up (see section VIRTUAL LINK REGISTER ONE OPERATION for further details).

1.2.18.13.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.18.14 VIRTUAL LINK DUMP PRIMITIVE

1.2.18.14.1 SYNOPSIS

This primitive dumps the content of a virtual link descriptor.

#include <vlx/vlink-lib.h>

int **vlink_dump** (Vlink∗ *vlink*);

1.2.18.14.2 DESCRIPTION

This primitive dumps the content of the virtual link descriptor using the Linux internal **printk** primitive. The virtual link library must be compiled in debug mode (*-DVLINK_DEBUG* option set) in order to get these traces.

1.2.18.14.3 PARAMETERS

The single *vlink* argument is a valid pointer to a *Vlink* object.

1.2.18.14.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.18.15 VIRTUAL LINK MACROS

1.2.18.15.1 SYNOPSIS

These macros are provided by the virtual link library for simple and quick operations.

static inline void **vlink_sessions_start** (Vlink∗ *vlink*);

1.2.18.15.2 DESCRIPTION

This macro starts sessions by setting **sessions_started** field to 1 and unlock the sessions start lock semaphore.

1.2.18.15.3 PARAMETERS

This macro is invoked with a valid pointer to a *Vlink* object.

1.2.18.15.4 RETURN VALUES

No relevant value is returned by this macro.

1.2.18.15.5 SYNOPSIS

static inline void **vlink_sessions_cancel** (Vlink∗ *vlink*);

1.2.18.15.6 DESCRIPTION

This macro cancels sessions. The current state of the virtual link must not be equal to *VLINK_CLEAN* (that is, not started).

1.2.18.15.7 PARAMETERS

This macro is invoked with a valid pointer to a *Vlink* object.

1.2.18.15.8 RETURN VALUES

No relevant value is returned by this macro.

1.2.18.15.9 SYNOPSIS

static inline void **vlink_session_enter** (VlinkSession∗ *session*);

1.2.18.15.10 DESCRIPTION

This macro is invoked to prevent a session to be deleted. The **users** field of the associated *Vlink* object is atomically incremented by 1.

1.2.18.15.11 PARAMETERS

This macro is invoked with a valid pointer to a *VlinkSession* object.

1.2.18.15.12 RETURN VALUES

No relevant value is returned by this macro.

1.2.18.15.13 SYNOPSIS

static inline void **vlink_session_leave** (VlinkSession∗ *session*);

1.2.18.15.14 DESCRIPTION

This macro is invoked to leave a session after an invocation of **vlink_session_enter(VlinkSession∗ s)** allowing a session to be deleted. The **users** field of the associated *Vlink* object is atomically decremented by 1.

1.2.18.15.15 PARAMETERS

This macro is invoked with a valid pointer to a *VlinkSession* object.

1.2.18.15.16 RETURN VALUES

No relevant value is returned by this macro.

1.2.18.15.17 SYNOPSIS

static inline void **vlink_session_destroy** (VlinkSession∗ *session*);

1.2.18.15.18 DESCRIPTION

This macro is invoked to destroy a session and free all memory allocated for this object. The session must not be in used (that is, *refcount* not equal to 1 after an atomic decrement and test operation). The **vlink_session_release(VlinkSession∗ s)** is invoked by this macro (see section VIRTUAL LINK SESSION RELEASE for further details).

1.2.18.15.19 PARAMETERS

This macro is invoked with a valid pointer to a *VlinkSession* object.

1.2.18.15.20 RETURN VALUES

No relevant value is returned by this macro.

1.2.18.15.21 SYNOPSIS

static inline int **vlink_session_enter_and_test_alive** (VlinkSession∗ *session*);

1.2.18.15.22 DESCRIPTION

This macro tests whether the session is alive (**state** field of the session descriptor set to *VLINK_SESSION_ALIVE*).

1.2.18.15.23 PARAMETERS

This macro is invoked with a valid pointer to a *VlinkSession* object.

1.2.18.15.24 RETURN VALUES

If the session is alive, 0 is returned, otherwise another non null value is returned (-1 for *VLINK_SESSION_NEW* and 1 for *VLINK_SESSION_ABORTED*).

1.2.18.15.25 SYNOPSIS

static inline void **vlink_session_get** (VlinkSession∗ *session*);

1.2.18.15.26 DESCRIPTION

This macro increments the reference count of a session allowing to prevent any release operation (**refcount** of a *VlinkSession* object).

1.2.18.15.27 PARAMETERS

This macro is invoked with a valid pointer to a *VlinkSession* object.

1.2.18.15.28 RETURN VALUES

No relevant value is returned by this macro.

1.2.18.15.29 SYNOPSIS

static inline void **vlink_session_put** (VlinkSession∗ *session*);

1.2.18.15.30 DESCRIPTION

This macro decrements the reference count of a session (**refcount** of a *VlinkSession* object). If the reference count reaches 0, the **vlink_session_release(VlinkSession∗ s)** is invoked by this macro (see section VIRTUAL LINK SESSION RELEASE for further details).

1.2.18.15.31 PARAMETERS

This macro is invoked with a valid pointer to a *VlinkSession* object.

1.2.18.15.32 RETURN VALUES

No relevant value is returned by this macro.

1.2.18.16 SEE ALSO

nk_xirq_attach

nk_xirq_detach

vrpq_be

vrpq_fe

1.2.19 VMQ(4D)

1.2.19.1 NAME

vmq — Virtual Message Queue Device driver interface layer for virtual drivers.

1.2.19.2 SYNOPSIS

#include <vlx-vmq.h>

This virtual Message Queue (VMQ) layer is used to provide a generic message library for virtual drivers.

1.2.19.3 FEATURES

This library is launched as a Linux module through the **insmod** command.

1.2.19.4 DESCRIPTION

The virtual Message Queue library provides a generic message queue interface for virtual drivers using virtual links to communicate across VMs. Each virtual link has a transmit and receive rings. This interface is composed of primitives managing messages, callback routines and data structures related to virtual links and configuration parameters.

1.2.19.5 EXTENDED DESCRIPTION

The VMQ interface is composed of two main objects: a *vmq_link_public_t* and a *vmq_xx_config_t* whose fields are shown below:

```
typedef struct vmq_link_t  vmq_link_t;
typedef struct vmq_links_t vmq_links_t;

typedef struct {
    void*        priv;       /* Must be first */
    NkOsId       local_osid;
    NkOsId       peer_osid;
    char*        rx_s_info;
    char*        tx_s_info;
    char*        rx_data_area;
    char*        tx_data_area;
    unsigned     data_max;
    unsigned     msg_max;
    NkPhAddr     ptx_data_area;
    NkPhAddr     prx_data_area;
    unsigned     prx_data_area_size;
} vmq_link_public_t;

typedef struct {
    unsigned     msg_count;
    unsigned     msg_max;
    unsigned     data_count;
    unsigned     data_max;
    vmq_pdev_level_t pdev_level;
} vmq_xx_config_t;
```

Both backend and frontend drivers are responsible for initializing a couple of *vmq_xx_config_t* objects for transmit and receive parameters. The backend driver is acting as a receiver and thus receive configuration parameters are provided by this driver. The frontend driver is acting as a transmitter and thus transmit configuration parameter are provided by this driver. The **msg_count** field is set to the number of messages to be transmitted or received. The **msg_max** is set to the size of each message. The **data_count** is set to the number of buffers for transmit or receive operations, and **data_max** is set to the size of each buffer.

A *vmq_links_t* object is provided to each client driver (that is, frontend and backend drivers), allocated by **vmq↵ _links_init()** at boot time (see section INITIALIZING VIRTUAL LINKS for further details). A *vmq_link_t* object is obtained per virtual link through the **vmq_links_iterate()** primitive (see sections EXECUTING A DEDICATED F↵ UNCTION ON MULTIPLE LINKS and GETTING INFORMATION ABOUT VIRTUAL LINKS for further details).

A *vmq_link_t* object is composed of a *vmq_link_public_t* as first field and is equivalent as a public derivation in C++.

The **priv** is devoted to virtual drivers (that is, clients of the VMQ driver) for their own use.

The **local_osid** is updated by the VMQ driver and is set to the VM identifier where the receiver is running.

The **peer_osid** is set to the VM identifier where the transmitter is running.

The **rx_s_info** can refer a string of characters of the receiver information.

The **tx_s_info** can refer a string of characters of the transmitter information.

The fields **tx_data_area** and **rx_data_area** respectively refer the start address of the transmit and receive ring buffers.

The fields **data_max** and **msg_max** are respectively set to the value of **data_max** and **msg_max** found in *vmq_xx_config_t* object and aligned on cache line boundaries. This latter object is provided by both backend and frontend drivers at initialization time (see **vmq_links_init()** for further details).

The fields **ptx_data_area** and **prx_data_area** respectively refer the physical start address of the transmit and receive ring buffers.

The field **prx_data_area_size** is set to the size of the receive ring.

The *vmq_xx_config_t* object contains also the **pdev_level** parameter which defines the layout of used persistent memory:

VMQ_PDEV_NONE

Nothing in pdev, everything in pmem

VMQ_PDEV_HEAD_INDEX

Head/index in pdev, short/long in pmem

VMQ_PDEV_SHORT

Head/index/short in pdev, long in pmem

VMQ_PDEV_LONG

Everything in pdev, nothing in pmem

The virtual Message Queue library exports the following API to virtual device drivers:

```
    /* Communication functions */
static signed
    vmq_msg_allocate    (vmq_link_t*, unsigned data_len, void** msg,
                 unsigned* data_offset) __must_check;
signed  vmq_msg_allocate_ex (vmq_link_t*, unsigned data_len, void** msg,
                 unsigned* data_offset, _Bool nonblocking)
                 __must_check;
signed  vmq_msg_allocate_many_data (vmq_link_t*, unsigned data_count,
                 void** msg, unsigned* data_offsets,
                 _Bool nonblocking);
void    vmq_msg_send        (vmq_link_t*, void* msg);
void    vmq_msg_send_async  (vmq_link_t*, void* msg);
void    vmq_msg_send_flush  (vmq_link_t*);
signed  vmq_msg_receive     (vmq_link_t*, void** msg) __must_check;
void    vmq_msg_free        (vmq_link_t*, void* msg);
void    vmq_msg_return      (vmq_link_t*, void* msg);
unsigned
    vmq_msg_slot        (vmq_link_t*, const void* msg) __must_check;
_Bool   vmq_data_offset_ok  (vmq_link_t*, unsigned data_offset)
                 __must_check;
void    vmq_data_free       (vmq_link_t*, unsigned data_offset);
void    vmq_data_free_many  (vmq_link_t*, unsigned data_count,
                 const unsigned* data_offsets);
signed  vmq_return_msg_receive  (vmq_link_t* link2, void** msg)
                 __must_check;
void    vmq_return_msg_free (vmq_link_t* link2, void* msg);

    /* Link control functions */
static signed
    vmq_links_init      (vmq_links_t**, const char* vlink_name,
                 const vmq_callbacks_t*,
                 const vmq_xx_config_t* tx_config,
                 const vmq_xx_config_t* rx_config)
                 __must_check;
signed  vmq_links_init_ex   (vmq_links_t**, const char* vlink_name,
                 const vmq_callbacks_t*,
                 const vmq_xx_config_t* tx_config,
                 const vmq_xx_config_t* rx_config, void* priv,
                 _Bool is_frontend) __must_check;
signed  vmq_links_start     (vmq_links_t* links);
void    vmq_links_finish    (vmq_links_t*);
_Bool   vmq_links_iterate   (vmq_links_t*, _Bool (*func)(vmq_link_t*,
                 void*), void* cookie);
void    vmq_links_sysconf   (vmq_links_t*);
void    vmq_links_abort     (vmq_links_t*);
```

All these primitives are explained in next sections. In addition, a set of macros is also provided to clarify client source code:

```
    static inline signed
vmq_msg_allocate (vmq_link_t* link2, unsigned data_len, void** msg,
          unsigned* data_offset)
{
    return vmq_msg_allocate_ex (link2, data_len, msg, data_offset,
                0 /*!nonblocking*/);
}

    static inline signed
vmq_links_init (vmq_links_t** links, const char* vlink_name,
        const vmq_callbacks_t* callbacks,
        const vmq_xx_config_t* tx_config,
        const vmq_xx_config_t* rx_config)
{
    return vmq_links_init_ex (links, vlink_name, callbacks, tx_config,
                  rx_config, NULL, false);
}

    static inline NkOsId
vmq_peer_osid (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->peer_osid;
```

```
}

    static inline const char*
vmq_link_rx_s_info (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->rx_s_info;
}

#define vmq_link_s_info vmq_link_rx_s_info

    static inline const char*
vmq_link_tx_s_info (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->tx_s_info;
}

    static inline char*
vmq_rx_data_area (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->rx_data_area;
}

    static inline char*
vmq_tx_data_area (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->tx_data_area;
}

    static inline unsigned
vmq_data_max (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->data_max;
}

    static inline unsigned
vmq_msg_max (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->msg_max;
}

    static inline NkPhAddr
vmq_ptx_data_area (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->ptx_data_area;
}

    static inline NkPhAddr
vmq_prx_data_area (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->prx_data_area;
}

    static inline unsigned
vmq_prx_data_area_size (const vmq_link_t* link2)
{
    return ((vmq_link_public_t*) link2)->prx_data_area_size;
}
```

A set of callbacks functions is also provided by the virtual Message Queue API for getting link status and to be woken up by notifications such as virtual link is on or off, receive message, and so on

```
    /*
     * Link control callbacks
     */
typedef struct {
    void (*link_on)      (vmq_link_t*);
    void (*link_off)         (vmq_link_t*);
    void (*link_off_completed)   (vmq_link_t*);
    void (*sysconf_notify)   (vmq_links_t*);
    void (*receive_notify)   (vmq_link_t*);
    void (*return_notify)    (vmq_link_t*);
```

```
    const vmq_xx_config_t*
     (*get_tx_config)    (vmq_link_t*, const char* tx_s_info);
    const vmq_xx_config_t*
     (*get_rx_config)    (vmq_link_t*, const char* rx_s_info);
} vmq_callbacks_t;
```

There is only one mandatory callback routine to provide by clients to the VMQ driver: *sysconf_notify*. This handler is invoked when a *NK_XIRQ_SYSCONF* cross interrupt is triggered by the VMQ driver for a reconfiguration event. The client must invoke **vmq_vlinks_sysconf()** primitive to get the related event. The section entitled GETTING INFORMATION ABOUT VIRTUAL LINKS describes how **link_on()**, **link_off()** and **link_off_completed()** callback routines are invoked by the VMQ driver. The **receive_notify()** callback invocation is explained in section REC←EIVING A MESSAGE FROM A RECEIVE RING, and **return_notify()** callback invocation is explained in section RECEIVING A MESSAGE FROM A TRANSMIT RING.

1.2.19.6 INITIALIZING VIRTUAL LINKS

1.2.19.6.1 SYNOPSIS

These primitive are invoked to initialize virtual links, callback functions and configuration.

#include <vlx-vmq.h>

signed **vmq_links_init** (vmq_links_t∗∗ *links*, const char∗ *vlink_name*, const vmq_callbacks_t∗ *callbacks*, const vmq_xx_config_t∗ *tx_config*, const vmq_xx_config_t∗ *rx_config*);

signed **vmq_links_init_ex** (vmq_links_t∗∗ *links*, const char∗ *vlink_name*, const vmq_callbacks_t∗ *callbacks*, const vmq_xx_config_t∗ *tx_config*, const vmq_xx_config_t∗ *rx_config*, void∗ *priv*, _Bool *is_frontend*);

1.2.19.6.2 DESCRIPTION

The **vmq_links_init()** primitive is invoked by both backend and frontend drivers to find and initialize virtual links between backend and frontend drivers, to set up callback functions and to record the appropriate configuration provided by the caller. The backend and frontend drivers respectively provide the receive and the transmit configuration parameters. This primitive must be invoked first when each client driver (that is, frontend and backend) is booting up.

A variant of this primitve is **vmq_links_init_ex()** which have two more parameters.

1.2.19.6.3 PARAMETERS

The first *links* argument must be a valid pointer to a vlink pointer which will be updated by this primitive and further used by all other primitives controlling links.

The second *vlink_name* argument is a pointer to a valid string of characters containing the name of the virtual link device previously recorded in platform device tree.

The third *callbacks* argument refers a valid *vmq_callbacks_t* object containing the entry point of all callback routines.

The fourth *tx_config* argument contains transmit parameters and must be filled with appropriate values provided by the frontend driver.

The fifth *rx_config* argument contains receive parameters and must be filled with appropriate values provided by the backend driver.

There are two more parameters for **vmq_links_init_ex()**:

The *priv* parameter, if non-null, gives a private value to the links.

The *is_frontend* is a boolean value that is true if the initialization is done in a frontend VM.

1.2.19.6.4 RETURN VALUES

In case of successful operation, a null integer is returned and the *vlink* pointer is updated accordingly to refer a valid *vmq_vlinks_t* object which will be used in all other primitives controlling virtual links. In case of error, the following error codes are returned:

EINVAL

is returned if the **sysconf_notify** field of the *callbacks* object is set to a null pointer. In other words, this callback routine is mandatory. This error code is also returned when the **msg_count** field of *rx_config* and/or *tx_config* objects is equal to zero or not a power of 2;

ENOMEM

is returned if the underlying data or cross interrupts cannot be allocated by the Hypervisor;

EAGAIN

is returned if the underlying *NK_XIRQ_SYSCONF* handler cannot be attached. This error code is also returned when the memory allocated for transmit or receive buffers cannot be mapped in the current VM virtual space.

1.2.19.7 RELEASING VIRTUAL LINKS

1.2.19.7.1 SYNOPSIS

This primitive releases all virtual links involved between frontend and backend drivers previously initialized by **vmq↩_links_init()**.

#include <vlx-vmq.h>

void **vmq_links_finish** (vmq_links_t∗ *links*);

1.2.19.7.2 DESCRIPTION

This primitive is called when either a client frontend and/or backend driver are going to shutdown. After this call the *links* argument is no longer valid since all underlying allocated data have been freed.

1.2.19.7.3 PARAMETERS

The single argument is a pointer on a valid *vmq_links_t* object. If this argument is a null pointer this primitive returns immediately (that is, a null argument is harmless).

1.2.19.7.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.19.8 EXECUTING A DEDICATED FUNCTION ON MULTIPLE LINKS

1.2.19.8.1 SYNOPSIS

This primitive allows to execute a function on a list of links.

#include <vlx-vmq.h>

_Bool **vmq_links_iterate** (vmq_links_t∗ *links*, _Bool∗ (*func*)(vmq_link_t∗, void∗), void∗ *cookie*);

1.2.19.8.2 DESCRIPTION

This primitive is invoked to execute a dedicated routine on each link belonging to the list of links referred by a *vmq_links_t* object. Both backend and frontend drivers must call **vmq_links_iterate()** to perform operations on virtual links. This primitive executes a loop on all the virtual links list belonging to a *vmq_links_t* object and invokes for each of them the dedicated function *func* through a non null pointer provided by the caller. At boot time, this primitive allows client drivers to record the *vmq_link_t* object associated with each virtual link.

1.2.19.8.3 PARAMETERS

The first *links* argument is a valid pointer to a *vmq_links_t* returned by **vmq_links_init()**.

The second *func* argument is a valid pointer to a dedicated function returning a boolean and having a couple of arguments whose the first is a pointer to a *vmq_link_t* object and the second is a pointer to a private data structure.

This latter parameter is given through the third *cookie* argument of **vmq_links_iterate()** primitive.

1.2.19.8.4 RETURN VALUES

This primitive returns 1 (**true**) if the dedicated function returns a non null value and 0 (**false**) when the loop is ended. Consequently, iterations on virtual links list stops until a dedicated function returns 1. If the dedicated function always returns 0, iterations on virtual links is executed until the end of list.

1.2.19.9 INFORMATION ABOUT VIRTUAL LINKS

1.2.19.9.1 SYNOPSIS

This primitive is invoked to get the current status of each virtual link belonging to a *vmq_links_t* object.

#include <vlx-vmq.h>

void **vmq_links_sysconf** (vmq_links_t∗ *links*);

1.2.19.9.2 DESCRIPTION

This primitive checks the current status and get information related to each virtual link belonging to a *vmq_links_t* object. This primitive is invoked by clients from *sysconf_notify* callback routine (that is, in the context of this callback) provided to VMQ driver when **vmq_links_init()** is invoked at initialization time. Three callback routines may be invoked by this primitive according to the current status of each virtual link:

link_off_completed()

callback routine is invoked when no message is pending in both receive and transmit rings of a virtual link and if one or both receive and transmit rings was aborted. The VMQ driver then checks if one or both rings are in *NK_DEV_VLINK_OFF* state, and in that case the *NK_DEV_VLINK_RESET* state is set. Note that this callback is invoked if the **link_off_completed** field belonging to the *vmq_callbacks_t* object previously given to **vmq_links_init()** is not a null pointer;

link_on()

callback routine is invoked when the virtual link status for both transmit and receive operations is set to *N*↩ *K_DEV_VLINK_ON*. In other words, both transmit and receive rings are ready for respectively sending and receiving messages. Note that this callback is invoked if the **link_on** field belonging to the *vmq_callbacks_t* object previously given to **vmq_links_init()** is not a null pointer;

link_off()

callback routine is invoked when one of the virtual link status for both transmit and receive operation is not set to *NK_DEV_VLINK_ON*. Note that this callback is invoked if the **link_off** field belonging to the *vmq_callbacks_t* object previously given to **vmq_links_init()** is not a null pointer.

For getting further details about virtual link protocol (transition state status: *NK_DEV_VLINK_OFF*, *NK_DEV_VL*↩ *INK_ON* and *NK_DEV_VLINK_RESET*) the **nk_vlink_lookup(3D)** manual page can be read.

1.2.19.9.3 PARAMETERS

The single *links* argument is a valid pointer to a *vmq_links_t* object previously updated by the **vmq_links_init()** primitive.

1.2.19.9.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.19.10 ABORTING VIRTUAL LINKS FOR TRANSMIT OPERATIONS

1.2.19.10.1 SYNOPSIS

This primitive is invoked to abort all virtual links for transmit operation belonging to a *vmq_links_t* object.

#include <vlx-vmq.h>

void **vmq_links_abort** (vmq_links_t∗ *links*);

1.2.19.10.2 DESCRIPTION

This primitive aborts all transmit virtual links belonging to a *vmq_links_t* object. The current status of each virtual link for transmit operations is set to "aborted" and all pending processes waiting for characters from these links are woken up. The **vmq_msg_allocate()** may be unblocked and return with an appropriate error code (see section ALLOCATING A MESSAGE for further details).

1.2.19.10.3 PARAMETERS

The single *links* argument is a valid pointer to a *vmq_links_t* object previously updated by the **vmq_links_init()** primitive.

1.2.19.10.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.19.11 ALLOCATING MESSAGES

1.2.19.11.1 SYNOPSIS

These primitives are invoked to allocate messages from a virtual link.

#include <vlx-vmq.h>

signed **vmq_msg_allocate** (vmq_link_t∗ *vlink*, unsigned *data_len*, void∗∗ *msg*, unsigned∗ *data_offset*);

signed **vmq_msg_allocate_ex** (vmq_link_t∗ *vlink*, unsigned *data_len*, void∗∗ *msg*, unsigned∗ *data_offset*, _Bool *nonblocking*)

signed **vmq_msg_allocate_many_data** (vmq_link_t∗ *vlink*, unsigned *data_count*, void∗∗ *msg*, unsigned∗ *data_↩ offsets*, _Bool *nonblocking*)

1.2.19.11.2 DESCRIPTION

The **vmq_msg_allocate()** primitive allocates a new message from the transmit ring of a given virtual link. When there is no more room in the transmit ring, this primitive may be blocked until memory is freed or this virtual link is reset (see previous section ABORTING VIRTUAL LINKS FOR TRANSMIT OPERATIONS for further details). Consequently, this primitive may block the caller for a non bounded time.

A first variant of this primitive, **vmq_msg_allocate_ex()**, allocates a message from a virtual link, with the option of blocking or not blocking the caller if the allocation fails.

A second variant of this primitive, **vmq_msg_allocate_many_data()**, allocates several messages in one request, with the option of blocking or not blocking the caller if the allocation fails.

1.2.19.11.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive.

The second *data_len* argument holds the required data associated with the newly allocated message. This value can be set to zero.

The third *msg* argument is a pointer to a pointer referring a private message (that is, data structure describing a message) defined by clients of the VMQ driver and allocated by this primitive.

The fourth argument *data_offset* is used in conjunction with **vmq_data_free()** primitive. When *data_len* is set to zero, this last argument may be a null pointer meaning that there is no data associated with the message. If *data_len* is not set to zero, *data_offset* must be a valid pointer in order to be updated by **vmq_msg_allocate()**. In that case **vmq_data_free()** must be invoked to free data associated with the message.

The fifth argument *nonblocking* (only for **vmq_msg_allocate_ex()** and **vmq_msg_allocate_many_data()**) can be used to enable a nonblocking behavior during the allocation.

1.2.19.11.4 RETURN VALUES

In case of success, this primitive returns a null value and *msg* is updated. If *data_offset* is not a null pointer, it is also updated according to the value of *data_len*. If this latter value is set to zero, *data_offset* is also set to zero if it is not a null pointer. If *data_len* is different from zero, *data_offset* is updated by this primitive, and must be used when the underlying data will be released through **vmq_data_free()** primitive. In case of error, the following negative error codes are returned:

EAGAIN

is returned if the current status of the virtual link associated with transmit operations is not equal to *NK_DEV↩ _VLINK_ON*. This means that the current virtual link is not ready to transmit due to a disconnection from either backend or frontend side, and is also returned if the *nonblocking* mode is true and the allocation failed;

ECONNABORTED

is returned if the current status of the virtual link associated with transmit operations is aborted;

EINTR

is returned when the caller has been blocked for getting a new message due to a transmit ring full, and has been woken up by a signal (Unix signal mechanism);

E2BIG

is returned if *data_len* is greater than the current value held in **data_max** field belonging to the *vmq_xx_config_t* object for transmit operation parameters and provided to **vmq_links_init()** primitive at initialization time by clients.

1.2.19.12 SENDING A MESSAGE TO A VIRTUAL LINK

1.2.19.12.1 SYNOPSIS

These primitives are invoked to send a message to a virtual link.

#include <vlx-vmq.h>

void **vmq_msg_send** (vmq_links_t∗ *vlink*, void∗ *msg*);

void **vmq_msg_send_async** (vmq_links_t∗ *vlink*, void∗ *msg*);

void **vmq_msg_send_flush** (vmq_links_t∗ *vlink*);

1.2.19.12.2 DESCRIPTION

The **vmq_msg_send()** primitive is invoked to send a message to a virtual link across a transmit ring. If the virtual link status of the transmit ring is in aborted state, the **sysconf_notify()** callback routine provided through a *vmq_callbacks_t* object to **vmq_links_init()** primitive at initialization time is invoked. In that case no message is sent. The **link_off** callback can also be invoked in case of link failure (that is, disconnection from either backend or frontend side). In all other cases, the message is sent from the virtual link transmit ring always in asynchronous mode. In other words, this routine never blocks the caller.

A first variant of this primitive, **vmq_msg_send_async()**, sends the message asynchronously, which means that the the vitual link is not flushed.

A second variant of this primitive, **vmq_msg_send_flush()**, explicitly flushes the virtual link.

1.2.19.12.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive. The second *msg* argument is a valid pointer to a client message which have been previously allocated through the **vmq_msg_allocate()** primitive.

1.2.19.12.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.19.13 RECEIVING A MESSAGE FROM A RECEIVE RING

1.2.19.13.1 SYNOPSIS

This primitive is invoked to receive a message from a virtual link.

#include <vlx-vmq.h>

void **vmq_msg_receive** (vmq_links_t∗ *vlink*, void∗∗ *msg*);

1.2.19.13.2 DESCRIPTION

This primitive allows to receive a message through the receive ring of a given virtual link. If the message is correctly received, the *msg* is updated by this primitive. This routine may be invoked in the context of the **receive_notify** callback routine allowing to asynchronously receive messages. This callback routine is provided to the VMQ driver by clients at initialization time through the **vmq_links_init()** primitive.

1.2.19.13.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive. The second *msg* argument is a valid pointer to a pointer. In case of success, this pointer is updated with the received message. At this stage the message is not consumed in the receiving ring. The **vmq_msg_free()** primitive must be invoked to free it.

1.2.19.13.4 RETURN VALUES

In case of success, this primitive returns a null value and *msg* refers a valid received message. In case of error, the following error codes are returned:

ESTALE

is returned if the message cannot be retrieved from the receive ring (invalid message count);

EAGAIN

is returned if no pending message is available from the receive ring.

This primitive is synchronous and never blocks to wait for a pending message. It is executed in the receiver context. In other words, this primitive is executed in a frontend driver for messages sent by a backend driver or conversely. In most cases, frontend and backend drivers are not supposed to run on the same VM.

1.2.19.14 RECEIVING A MESSAGE FROM A TRANSMIT RING

1.2.19.14.1 SYNOPSIS

This primitive is invoked to reply to a receiving message.

#include <vlx-vmq.h>

signed **vmq_return_msg_receive** (vmq_links_t$*$ *vlink*, void$**$ *msg*);

1.2.19.14.2 DESCRIPTION

This primitive replies to a receiving message through the transmit ring of a given virtual link in the context of the sender. In other words, this primitive is executed in the context of the **return_notify** callback. This callback routine is provided by clients when the **vmq_links_init()** primitive is invoked at initialization time. If the message is correctly received, the *msg* pointer is updated by this primitive. Then, the message can be released by invoking the **vmq_return_msg_receive** primitive always in the context of **return_notify** callback (see section RELEASING A MESSAGE AT VIRTUAL INTERRUPT CONTEXT for further details).

1.2.19.14.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive.

The second *msg* argument is a valid pointer to a pointer. In case of success, this pointer is updated with the received message. At this stage the message is not consumed in the receiving ring. The **vmq_return_msg_free()** primitive must be invoked to free it.

1.2.19.14.4 RETURN VALUES

In case of success, 0 is returned otherwise a negative error code is returned. The list of possible error codes is described below:

ESTALE

is returned if the message cannot be retrieved from the transmit ring (invalid message count);

EAGAIN

is returned if no pending message is available from the transmit ring.

As mentioned for **vmq_msg_receive()**, this primitive is synchronous and never blocks to wait for a pending message. It is always executed in the sender context as indicated above.

1.2.19.15 RELEASING A MESSAGE

1.2.19.15.1 SYNOPSIS

These primitives release a previously received message.

#include <vlx-vmq.h>

void **vmq_msg_free** (vmq_links_t∗ *vlink*, void∗ *msg*);

void **vmq_msg_return** (vmq_links_t∗ *vlink*, void∗ *msg*);

1.2.19.15.2 DESCRIPTION

The primitives release a previously received message from a receive ring belonging to a virtual link. If the receiving ring of the virtual link is in the aborted state, either the **link_off()** or **sysconf_notify()** callback routines belonging to clients is invoked by the VMQ driver. These callback routines are recorded when **vmq_links_init()** is invoked at initialization time. If the receive ring is full, all waiting clients are woken up.

The difference between this couple of primitives is that a cross interrupt is always sent to the producer (that is, to transmit ring of the message sender) when **vmq_msg_return()** is invoked. The same cross interrupt is sent when **vmq_msg_free()** is invoked if and only if the receive ring of the virtual link is full (that is, transmit ring of the message sender).

1.2.19.15.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive.

The second *msg* argument refers the message to release.

1.2.19.15.4 RETURN VALUES

No relevant value is returned by these primitives.

1.2.19.16 RELEASING DATA ASSOCIATED WITH MESSAGES

1.2.19.16.1 SYNOPSIS

These primitive release data associated with messages.

#include <vlx-vmq.h>

void **vmq_data_free** (vmq_links_t∗ *vlink*, unsigned *data_offset*);

void **vmq_data_free_many** (vmq_links_t∗ *vlink*, unsigned *data_count*, const unsigned∗ *data_offsets*);

1.2.19.16.2 DESCRIPTION

The **vmq_data_free()** primitive releases data associated with a previously sent message. As mentioned in section ALLOCATING A MESSAGE, if *data_len* is not set to zero, *data_offset* is filled by **vmq_msg_allocate()** to a non null value in case of success. So, in that case, the **vmq_data_free()** primitive is invoked to release associated data.

The **vmq_data_free_many()** primitive releases data associated previous sent messages. As mentioned in section ALLOCATING A MESSAGE, if *data_count* is not set to zero, *data_offsets* is filled by **vmq_msg_allocate_many←_data()** to a non null value in case of success. So, in that case, the **vmq_data_free_many()** primitive is invoked to release associated data.

1.2.19.16.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive.

The second *data_offset* argument holds the value returned by **vmq_msg_allocate()**.

1.2.19.16.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.19.17 RELEASING A MESSAGE AT VIRTUAL INTERRUPT CONTEXT

1.2.19.17.1 SYNOPSIS

This primitive releases a message from a transmit ring of a virtual link in the context of *return_notify* callback routine.

#include <vlx-vmq.h>

void **vmq_return_msg_free** (vmq_links_t∗ *vlink*, void∗ *msg*);

1.2.19.17.2 DESCRIPTION

This primitive is called by a client of the VMQ driver in the context of *return_notify* callback routine is invoked. In other words, when the VMQ drivers invokes the *return_notify* callback, previously recorded by **vmq_vlinks_init()** invocation at initialization time, the **vmq_return_msg_free()** can be invoked to a free a message from the transmit ring of a virtual link. This primitive allows to asynchronously free messages from a transmit ring.

1.2.19.17.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive.

The second *msg* argument is a valid pointer to a client message which have been previously allocated through the **vmq_return_msg_receive()** primitive (see section RECEIVING A MESSAGE FROM A TRANSMIT RING for further details). If the transmit ring is in the aborted state and there is at least one pending message in the transmit ring, the **sysconf_notify** callback routine of the client is invoked by **vmq_return_msg_free()**.

1.2.19.17.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.19.18 TESTING DATA OFFSET FOR RECEIVING MESSAGES

1.2.19.18.1 SYNOPSIS

This primitive checks the validity of a data offset in the receive queue.

#include <vlx-vmq.h>

_Bool **vmq_data_offset_ok** (vmq_links_t∗ *vlink*, unsigned *data_offset*);

1.2.19.18.2 DESCRIPTION

This primitive compares a given data offset against the total size of the receive queue memory given by **data_count** which is the total number of buffers of this receive queue multiplied by the size of each buffer *data_max* (see section EXTENDED DESCRIPTION for further details). This couple of values are provided by clients at initialization time when **vmq_links_init()** is invoked (that is, receive parameters are given by the backend, transmit parameters are given by the frontend).

1.2.19.18.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive. The second *data_offset* argument is an offset in the receive queue expressed in bytes.

1.2.19.18.4 RETURN VALUES

This primitive returns 1 (true) when *data_offset* is less than the total size of the receive queue, otherwise 0 (false) is returned.

1.2.19.19 INTERFACE FOR GETTING PUBLIC LINK INFORMATION

1.2.19.19.1 SYNOPSIS

These macros are provided to clarify the client source code.

#include <vlx-vmq.h>

static inline NkOsId **vmq_peer_osid** (const vmq_links_t∗ *vlink*);

static inline const char∗ **vmq_link_s_info** (const vmq_links_t∗ *vlink*);

static inline char∗ **vmq_tx_data_area** (const vmq_links_t∗ *vlink*);

static inline char∗ **vmq_rx_data_area** (const vmq_links_t∗ *vlink*);

static inline unsigned **vmq_data_max** (const vmq_links_t∗ *vlink*);

static inline unsigned **vmq_msg_max** (const vmq_links_t∗ *vlink*);

1.2.19.19.2 DESCRIPTION

The following services are provided:

vmq_peer_osid()

returns the guest OS identifier where the transmitter (that is, front—end driver) is running;

vmq_link_s_info()

returns the string of characters containing the receiver information (see section EXTENDED DESCRIPTION) for further details;

vmq_rx_data_area()

returns the start address of the receive queue;

vmq_tx_data_area()

returns the start address of the transmit queue;

vmq_data_max()

returns the size of each buffer in either a transmit or a receive queue;

vmq_msg_max()

returns the total number of messages of either a transmit or a receive queue;

1.2.19.19.3 PARAMETERS

The first *vlink* argument is a valid pointer on a *vmq_link_t* object previously got through the **link_on()** callback routine (invoked when both receive and transmit rings are ready for a given virtual link), or from the dedicated function invoked by **vmq_links_iterate()** primitive.

1.2.19.19.4 RETURN VALUES

No error code is returned by these macros, only relevant information according to their respective meaning. An unsigned integer for a guest identifier, a constant string of characters for receiver information, a constant pointer to a character for the couple of starting addresses of receive and transmit queues, and unsigned integers for the size of receive/transmit buffer and their related number of messages.

1.2.19.20 SEE ALSO

nk_mem_map

nk_pmem_alloc

nk_ptov

nk_pxirq_alloc

nk_vtop

nk_xirq_attach

nk_xirq_detach

nk_xirq_trigger

1.2.20 VPD(4D)

1.2.20.1 Cross References

Related Documents
Manual Page

1.2.20.2 NAME

vpd — Virtual Power Domain Drivers

1.2.20.3 SYNOPSIS

The virtual power domain (vpd) back-end and front-end drivers, when paired together, allow Linux power domain consumers defined in one VM to reference Linux power domain providers defined in another VM.

1.2.20.4 FEATURES

Under Linux, power domains are typically represented by nodes in the device tree. Those nodes are called **power domain providers**. Devices that are part of those power domains are called **power domain consumers**. For a general overview of those concepts, please refer to `Linux power domain device tree binding.`

Virtual power domain back-end driver's responsability is to export local power domain providers to other VMs. It can be enabled in the Linux kernel configuration:

Device Drivers -> VLX virtual device support -> VLX power domain back end driver

Virtual power domain front-end driver's responsability is to allow local power domain consumers to reference power domain providers exported by other VMs. It can be enabled in the Linux kernel configuration:

Device Drivers -> VLX virtual device support -> VLX power domain front end driver

Virtual power domain back-ends and front-ends communicate using VRPC. One VRPC link must be declared in the platform device tree for each (back-end, front-end) pair that shares power domains. The example below declares one vrpc link (VM2, VM3) using the virtual link framework (see vrpc manual page for more details).

```
&vm2_vdevs {
   vpd_be: vpd@be {                  // vpd back-end
        compatible = "vrpc";         // uses generic vRPC protocol
        server;                      //
        info       = "vpd_ctrl";     // driver name
   };
};

&vm3_vdevs {
    vpd@fe {                         // VM3 vpd front-end
        peer-phandle = <&vpd_be>;    // peer vLINK
        client;                      // front-end end point
        info         = "vpd_ctrl";   // driver name
    };
};
```

Below is a typical example of a power domain consumer and a power domain provider:

```
// Native Linux DTS.

    // power domain provider
    my_pd_provider: pd_provider {
        compatible = "vendor,some-pd-controller";
        #power-domain-cells = <2>;
    };

    // power domain consumer
    my_pd_consumer: devx {
        compatible = "vendor,some-dev-controller";
        power-domains = <&my_pd_provider 433 26>;
        power-domain-names = "mainpd";
    };
```

Below example shows how above configuration can be modified to move power domain consumer to another VM.

```
// VM2 Linux DTS configuring back-end.

    // power domain provider
    my_pd_provider: pd_provider {
        compatible = "vendor,some-pd-controller";
        #power-domain-cells = <2>;
    };

    // power domain consumer: disabled, but kept for parsing
    my_pd_consumer: devx {
        compatible = "disabled";
        power-domains = <&my_pd_provider 433 26>;
        power-domain-names = "mainpd";
    };
```

```
    /*
      Node describing power domains that are exported to VM3.
      Device tree may contain multiple "vl,vpower-domain-be" compatible nodes,
      one per peer VM.
    */
    vlx-pd-be@3 {

                compatible = "vl,vpower-domain-be";

                // This node describes power domains exported to VM3.
                vl,vm-id = <3>;

                // We export one power domain provider to VM3.
                vl,power-domain-providers = <&my_pd_provider>;

                // Names identifying the exported providers. Those will
                // be used by vpd front-end in VM3.
                vl,power-domain-provider-names = "my_pd_provider";

                // We only export the power domain used by one or several
                // power domain consumers.
                // Note that if an exported provider has a property
                // #power-domain-cell-size=0, then its unique domain is
                // always exported and doesn't require the following nodes.
                pd@0 {
                    vl,power-domain-ref = <&my_pd_consumer>;

                    // You could further limit the export to power domains
                    // specified by names, adding this optional property.
                    // vl,power-domain-names = "mainpd";

                    // You could force power-on previously named domains using
                    // this property. Note: if CONFIG_VLX_PM_ENABLE_ALL is defined,
                    // this property is ignored and all target power domains are
                    // force powered-on.
                    // vl,power-domain-init-on = <0>, <1>;
                };

                // You could add other consumers here.
                // pd@1 {
                //    vl,power-domain-ref = <&my_other_pd_consumer>;
                // };

    };
```

Next step is to configure a virtual power domain front-end in VM3 to access the exported power domains, as seen in below example:

```
// VM3 Linux DTS configuring front-end.

    // power domain provider
    my_pd_provider: pd_provider {
        compatible = "vl,vpower-domain-fe";
        vl,power-domain-provider-name = "my_pd_provider";
        #power-domain-cells = <2>;
    };

    // power domain consumer
    my_pd_consumer: devx {
        compatible = "vendor,some-dev-controller";
        power-domains = <&my_pd_provider 433 26>;
        power-domain-names = "mainpd";
    };
```

In the above example, power domain consumer is defined as usual, but power domain providers have to rely on device tree binding "vl,vpower-domain-fe". Each power domain provider must include a property "vl,power-domain-provider-name" set to back-end provider name, as it was specified by "vl,power-domain-provider-names" in the back-end device tree.

1.2.20.5 NOTES

An extra option in Linux kernel configuration allows you to always enable all power domains identified by the back-end driver at boot time:

Device Drivers -> VLX virtual device support -> Power on all exported power domains at boot time

1.2.20.6 SEE ALSO

vrpc

`Device tree bindings for power domain providers and power domain consumers.`

1.2.21 VPIPE(4D)

1.2.21.1 NAME

vpipe — Virtual Pipe

1.2.21.2 DESCRIPTION

The Virtual Pipe feature provides a communication link between Linux user space applications running in two different Virtual Machines. The semantics provided is identical to that of a Unix pipe.

The service is provided by a Linux kernel module. Each vpipe driver acts simultaneously as a front-end and as a back-end driver. It relies on the Hypervisor vlink service

1.2.21.3 CONFIGURATION

1.2.21.3.1 Linux Kernel Configuration

The virtual pipe driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual pipe for inter OS communication

It can be compiled as a loadable module or as an embedded driver.

1.2.21.3.2 Device Tree Configuration

Virtual pipe devices must be declared in the platform device tree. The example below declares vpipe devices using the virtual link framework(see the nk_vlink_lookup manual page for more details).

```
&vm2_vdevs {

    vpipe_0: vpipe@0 {          // vPIPE VM2 <- VM3
        compatible = "vpipe";   //
        link       = <0>;       //
        server;                 // server end point
    };
}

&vm3_vdevs {
    vpipe@0 {                         // vPIPE VM3 -> VM2
        peer-phandle = <&vpipe_0>;    // peer vLINK
        client;                       // client end point
    };
}
```

The vpipe feature provides a point to point communication link with one producer (client) and one consumer (server). vpipe devices appear as character devices in the Linux file system. By default, such files are names /dev/vpipeX where X is an automatically generated number.

An info field in the server device tree node may be used to change the default size and name:

```
    info = [<size>][;[name=<user_selected_name>]]
```

A info field in the client device tree node may be used to change the default name:

```
     info = [name=<user_selected_name>]
```

In such a case, the file will appear as /dev/user_selected_name. In case the user selected name appears more than once, it is ignored on the second occurrence and the second vpipe name is reverted to its default value (/dev/vpipeX).

1.2.21.4 USER SPACE DESCRIPTION

vpipes appear as character devices in the Linux filesystem. The nodes are created by the vpipe driver. A Linux user space application can then use regular file system calls such as open(2), read(2), Write(2) and close(2) to receive or send data through the pipe to another user space application running in a possibly different Virtual Machine.

1.2.21.5 /proc/nk Entries

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

```
      Mi Pr Id EDU St Size Opns Reads ReadBytes- Wrtes WriteBytes Name
```

There is one line printed for each vpipe managed by the Linux kernel. The vpipe device name appears as the right-most column.

Parameters

Mi	minor device number in the Linux filesystem
Pr	peer Virtual Machine id
Id	vlink's unique link id, as there can be several vpipes between a pair of virtual machines
EDU	"E" stands for Enabled "D" is set to "R" if the entry is for the server (consumer) side and "W" if the entry is for the client (producer) side "U" is current open count, usually 0 or 1, though it can be higher if several processes read from or write to vpipe
St	state of the underlying vlink and its two end-points: client side 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.

1.2.21.6 KERNEL DESCRIPTION

1.2.21.6.1 SYNOPSIS

#include <nk/nkern.h>

The virtual pipe driver runs on the top of the Linux kernel. This driver is given as an example, and can be adapted to run on the top of realtime OS-es. The virtual pipe mimics the behavior of traditional Unix/POSIX unidirectional pipes. The same vpipe driver can act as both frontend (writer side) or backend (reader side) driver.

1.2.21.6.2 OPENING A PIPE

1.2.21.6.2.1 SYNOPSIS

Before using a virtual pipe device, it must be opened by invoking the **ex_open** primitive. This operation is executed by the Linux kernel when an application performs **open** system call.

unsigned int **ex_open** (struct inode∗ *inode*, struct file∗ *file*);

1.2.21.6.3 DESCRIPTION

The **ex_open** primitive is responsible to open a virtual pipe. Upon the first open operation, the **ex_xirq_hdl** cross interrupt handler is attached. At the same time time a synchronization (handshake) with the peer driver is done through the Linux kernel primitive **wait_event_freezable**. Multiple open operations are allowed on a same virtual pipe (that is, same couple of *inode*, *file* parameters). In that case, the internal counter of *ExDev* object is incremented, no additional synchronization with peer driver is performed. The peer driver synchronization is implemented in the internal **ex_link_ready**) function. It consists in starting up the virtual link handshake protocol in order to establish communication between the backend and its peer counterpart (that is, the frontend driver) (see nk_vlink_lookup manual page for further details).

1.2.21.6.3.1 PARAMETERS

The *inode* parameter is a valid pointer to a Unix inode (system data part associated with a file) defined in **linux/fs.h**.

The *file* parameter is a pointer to a file data structure also defined in **linux/fs.h** file containing all data related to a file descriptor.

1.2.21.6.3.2 RETURN VALUES

In case of success, this primitive returns 0 otherwise the following error codes are returned:

ENXIO

is returned in case of incorrect minor number extracted from the caller's *inode* parameter, or when the underlying initialization (**ex_dev_init**) has failed;

EACCESS

is returned in case of incorrect access rights (that is, not readonly for a server and/or not writeonly for a client);

ENOMEM

is returned if the **ex_xirq_hdl** cross interrupt handler cannot be connected. This error code is returned only upon the first open operating on the virtual pipe;

EINTR

is returned if frontend/backend driver synchronization is interrupted by a signal;

1.2.21.6.4 RELEASING OR CLOSING A PIPE

1.2.21.6.4.1 SYNOPSIS

A virtual pipe is shutdown or closed when **ex_release** is invoked. This operation is executed by the Linux kernel when an application performs **close** or **exit** system calls.

unsigned int **ex_release** (struct inode∗ *inode*, struct file∗ *file*);

1.2.21.6.4.2 DESCRIPTION

The **ex_release** implements a close or a release operation required by Linux semantic for character devices. In case of read/write errors (**ex_read**, **ex_write**) an application can execute **close** system call in order to shut the link (virtual pipe) down and to set the link state to *NK_DEV_VLINK_OFF*. As **ex_open** can be invoked multiple times on a same virtual pipe, the close operation is effectively done when the counter described in *ExDev* object reaches zero.

When the counter reaches zero, the **ex_xirq_hdl** is detached through **ex_dev_cleanup** (see nk_xirq_detach manual page for further details) and the status of the current mode: client or server is set to *NK_DEV_VLINK_OFF*. Finally a *NK_XIRQ_SYSCONF* is triggered according to the current side of communication: client or server through the nk_xirq_trigger primitive in order to close respectively the server and the client side of the virtual pipe. The **ex_sysconf_trigger** subroutine is responsible for triggering the *NK_XIRQ_SYSCONF* cross interrupt to inform the peer driver that the state of the vlink is changed. The **ex_handshake** function is responsible for processing this cross interrupt.

1.2.21.6.4.3 PARAMETERS

The *inode* parameter is a valid pointer to a Unix inode (system data part associated with a file) defined in **linux/fs.h**.

The *file* parameter is a pointer to a file data structure also defined in **linux/fs.h** file containing all data related to a file descriptor.

1.2.21.6.4.4 RETURN VALUES

In case of success, this primitive returns 0, otherwise *EINTR* is returned if this operation is aborted because of signal while waiting on a mutes protecting the corresponding *ExDev* data structure.

1.2.21.6.5 READING FROM A PIPE

1.2.21.6.5.1 SYNOPSIS

Characters are read from a virtual pipe using **ex_read** primitive. This primitive is invoked from the server side of a virtual pipe. This operation is executed by the Linux kernel when an application performs **read** system call.

unsigned int **ex_read** (struct file∗ *file*, char __user∗ *buf*, size_t *count*, loff_t∗ *ppos*);

1.2.21.6.5.2 DESCRIPTION

This primitive implements a read operation required by Unix semantic for character devices. The **ex_read** attempts to read all *count* bytes from the writer side (client). If the underlying circular buffer is empty, this routine waits for characters if **ex_open** has been called without the bit *O_NONBLOCK* set in **file->f_flags** (see the Linux/Unix manual page of POSIX **open** for further details). If the circular buffer becomes non full after **ex_read**, a cross interrupt is sent to the client side in order to continue write operations.

In case of non blocking I/O (*O_NONBLOCK* bit is set), the caller is never blocked. The **ex_read** returns the number of characters read from the circular buffer. If the circular buffer was empty **ex_read** returns *EAGAIN*.

1.2.21.6.5.3 PARAMETERS

The *file* parameter is a valid pointer to a file data structure defined in **linux/fs.h** file containing all data related to a file descriptor.

The *buf* parameter is a valid pointer in user space (the *__user* macro is defined in file **linux/compiler.h** indicating to the C GNU compiler that the pointer is belonging to the user virtual space) for storing characters.

The *count* parameter is the number of required bytes to read and *ppos* is not used in this driver.

1.2.21.6.5.4 RETURN VALUES

In case of success, the number of read bytes is returned. This number can be equal or less that the required *count* given by the caller. In case of error, the following error codes are returned:

EBADF

is returned in case of client access, as mentioned above, only server side is allowed to read from a virtual pipe;

EAGAIN

is returned if no characters have been read while the current link status is still alive (that is, equal to *NK_DE↩ V_VLINK_ON*) and the read is in non—blocking mode (bit *O_NONBLOCK* set to one);

EFAULT

is returned if the *buf* is an invalid user address;

EINTR

is returned when current read operation is aborted because of signal.

1.2.21.6.6 WRITING TO A PIPE

1.2.21.6.6.1 SYNOPSIS

Characters are written to a virtual pipe using **ex_write** primitive. This primitive is invoked from the client side of a virtual pipe. This operation is executed by the Linux kernel when an application performs a **write** system call.

unsigned int **ex_write** (struct file∗ *file*, char __user∗ *buf*, size_t *count*, loff_t∗ *ppos*);

1.2.21.6.6.2 DESCRIPTION

This primitive implements a write operation required by Unix semantic for character devices. The **ex_write** transfers all *count* bytes to the reader side (server). If the underlying circular buffer is full, this routine waits until there is some room to store the required characters. If the circular buffer becomes non empty, a cross interrupt is sent to the server side in order to continue read operations.

In the case of non blocking I/O (*O_NONBLOCK* bit is set), the caller is never blocked. The "small" writes (with the size less or equal to the size of circular buffer) are never partial: if there is enough room in the circular buffer, all bytes would be written, otherwise no bytes would be written at all and **ex_write** returns *EAGAIN*. The "big" writes are always partial. The **ex_write** returns how many bytes were successfully written. If the circular buffer is full **ex_write** returns *EAGAIN*.

1.2.21.6.6.3 PARAMETERS

The *file* parameter is a valid pointer to a file data structure defined in **linux/fs.h** file containing all data related to a file descriptor.

The *buf* parameter is a valid pointer in user space (the *__user* macro is defined in file **linux/compiler.h** indicating to the C GNU compiler that the pointer is belonging to the user virtual space) for storing characters.

The *count* parameter is the number of required bytes to write and *ppos* is not used in this driver.

1.2.21.6.6.4 RETURN VALUES

In case of success, the returned value is equal to the required *count* given by the caller. In case of error, the following error codes are returned:

EPIPE

is returned if the current state of the server side is no longer equal to *NK_DEV_VLINK_ON*;

EAGAIN

is returned if the device has been opened with *O_NONBLOCK* bit set and if there is no enough room in the underlying circular buffer for "small" writes (size less or equal to the circular buffer size) or if the circular buffer is full for "big" writes;

EFAULT

is returned if the *buf* is an invalid user address;

EINTR

is returned when current write operation is aborted because of signal&.

1.2.21.6.7 POLLING FROM A PIPE

1.2.21.6.7.1 SYNOPSIS

The **ex_poll** checks if any characters can be read or if there is any room to write characters to a virtual pipe. This primitive is invoked from the client side of a virtual pipe. This operation is executed by the Linux kernel when an application performs system calls like **select**.

unsigned int **ex_poll** (struct file∗ *file*, poll_table∗ *wait*);

1.2.21.6.7.2 DESCRIPTION

This primitive implements a poll operation required by Unix semantic for character devices. It is responsible to check if there are any characters to read from the virtual pipe, or if there is any room to write characters to the virtual pipe. Consequently, this primitive can be called from both side: client or server and the result is returned accordingly.

1.2.21.6.7.3 PARAMETERS

The *file* parameter is a valid pointer to a file data structure defined in **linux/fs.h** file containing all related to a file descriptor.

The *wait* parameter is a valid pointer to a *poll_table* object defined in **linux/poll.h** file. This data structure is used by the kernel Linux primitive **poll_wait** also defined in the same file.

1.2.21.6.7.4 RETURN VALUES

A zero value is returned from the client side if there is no characters to read from the virtual pipe, or from the server side if there is no room to write at least one character to the virtual pipe.

A non zero value is returned on the client side if at least one character can be read from the virtual pipe. In that case, the error code holds two bits set *POLLIN* and *POLLRDNORM*. This couple of bits are defined in **linux/asm/poll.h** file.

A non zero value is returned on the server side if at least one character can be written to the virtual pipe. In that case, the error code holds two bits set *POLLOUT* and *POLLWRNORM*. This couple of bits are defined in **linux/asm/poll.h** file.

1.2.21.7 IMPLEMENTATION

1.2.21.7.1 VIRTUAL LINKS

The generic unidirectional point-to-point (P2P) virtual communication link is represented by the following data structures visible for both virtual backend and frontend drivers:

```
@#define NK_DEV_VLINK_OFF   0
@#define NK_DEV_VLINK_RESET 1
@#define NK_DEV_VLINK_ON    2

@#define NK_DEV_VLINK_NAME_LIMIT    16

typedef struct NkDevVlink {
    char        name[NK_DEV_VLINK_NAME_LIMIT];
                                /* name of communication link */
                                /* (actually virtual device class, */
                                /* like veth, vbd, etc.) */
```

```
                                /* max 15 characters + ending zero */
    int             link;       /* global/unique communication link number */
    NkOsId          s_id;       /* server OS id */
    volatile int    s_state;    /* server OS state: off, reset, on */
    nku32_f         s_info;     /* server specific info */
    NkOsId          c_id;       /* client OS id */
    volatile int    c_state;    /* client OS state: off, reset, on */
    nku32_f         c_info;     /* client specific info */
    nku32_f         pad0;
    nku64_f         pad1;
} NkDevVlink;
```

These links are used between two VMs. The **name** field is the name of the underlying virtual device such as veth, vpipe, vbpipe, vupipe and so on. The **link** field is a global and unique number differentiating virtual links belonging to the same virtual device class. The fields **s_id** and **c_id** are respectively the server VM identifier over which the backend driver is running and the client VM identifier over which the frontend driver is running.

The fields **s_state** and **c_state** respectively hold the server state and the client state which can take the following values: *NK_DEV_VLINK_OFF* the virtual link is turned off and is not initialized or must be reinitialized, *NK_DEV↩_VLINK_ON* the virtual link is turned on and ready to communicate with its counterpart (respectively frontend and backend drivers) and finally *NK_DEV_VLINK_RESET* the virtual link initialization is completed and not yet activated. The nk_vlink_lookup manual page contains further details about virtual links. In addition, this manual page provides a more detailed description about the handshake protocol between a backend and frontend driver.

The couple of fields **s_info** and **c_info** hold respectively a string of characters containing device specific information about server and client sides.

1.2.21.7.2 EXTENDED DESCRIPTION

The virtual pipe device is implemented as a simple circular buffer of characters using a couple of free running indexes for the producer which is writing characters to this buffer, and for the consumer which is reading characters from this buffer. Cross interrupts are sent to alert each peer driver (producer and consumer) when that circular buffer became either non empty or non full.

By convention a driver connected to a client side of a communication link (it is also called frontend driver) puts/writes characters into this circular buffer, and a driver connected to a server side of a communication link (it is also called backend driver) gets/reads characters from this circular buffer.

This circular buffer (also called ring buffer) is located in the shared persistent memory (see nk_pmem_alloc and nk_mem_map manual pages for further details), thus visible to both sides of the link. The layout of this buffer is a *ExRing* object and has the following layout:

```
@#define DEF_RING_SIZE      0x1000
@#define MIN_RING_SIZE          24

typedef struct ExRing {
    volatile nku32_f  s_idx;       /* "Free running" "server" index */
    volatile nku32_f  c_idx;       /* "Free running" "client" index */
    nku8_f    ring[MIN_RING_SIZE]; /* Circular communication buffer */
} ExRing;
```

The **s_idx** variable is a "free running" server index. It is incremented by the backend driver (server) each time when it reads characters from the circular buffer. It is never decremented. In order to use it as a buffer index, it should be get modulo buffer size. The **c_idx** variable is a "free running" client index. It is incremented by the frontend driver (client) each time when it writes characters to the circular buffer.

When we run several VMs on top of the Hypervisor, there is no synchronization between VMs. In addition, the Hypervisor can switch to another VM at any moment. The fields **s_idx** and **c_idx** have a volatile attribute since they can be changed by a peer driver at any moment, so the C compiler is not allowed to optimize these fields accesses.

The virtual pipe driver uses cross interrupts mechanism provided by the Hypervisor device driver framework in order to wake its peer driver up when some work must be done either to read some characters from the link by the consumer (because the producer has written some characters to it), or to write some characters to the link by the producer (because the consumer has read some characters from it).

Macros are provided to manage the available room in the ring buffer from both the consumer and the producer side and are described below:

```
    /*
     * We use the following macros to calculate available space ("room")
     * in the circular buffer (see 2 diagrams below)
     *
     *          s_idx               c_idx
     *            |      available     |
     *            |    consumer room   |
     *            |<------------------>|
     *            |                    |
     *  |---------v--------------------v----------------------|
     *            |                                           |
     *            |<----------------------------------------->|
     *                       contiguous consumer room
     *
     *
     *          s_idx               c_idx
     *            |                    |       available
     *            |                    |     producer room
     *  --------->|                    |<----------------------
     *            |                    |
     *  |---------v--------------------v----------------------|
     *                                 |                      |
     *                                 |<-------------------->|
     *                                          contiguous
     *                                        producer room
     *
     *
     * RING_P_ROOM  - how much "room" in a circular ring (i.e. how many
     *           available bytes) we have for the producer
     * RING_P_CROOM - how much "contiguous room" (from the current position
     *           up to end of ring w/o ring overlapping)
     *           in a circular ring we have for the producer
     * RING_C_ROOM  - how much "room" in a circular ring (i.e. how many
     *           available bytes) we have for the consumer
     * RING_C_CROOM - how much "contiguous room" (from the current position
     *           up to end of ring w/o ring overlapping)
     *           in a circular ring we have for the consumer
     */
#define RING_P_ROOM(rng,size)    ((size) - ((rng)->c_idx - (rng)->s_idx))
#define RING_P_CROOM(ex_dev)     ((ex_dev)->size - (ex_dev)->pos)
#define RING_C_ROOM(rng)     ((rng)->c_idx - (rng)->s_idx)
#define RING_C_CROOM(ex_dev)     ((ex_dev)->size - (ex_dev)->pos)
```

For each vpipe device the virtual pipe driver has a private (not visible by its peer driver) data structure describing this device. It is allocated and initialized when the device driver is loaded as a module or when the Linux kernel is booted if the driver is compiled as an embedded one. The layout of a *ExDev* whose members are described below:

```
typedef struct ExDev {
    _Bool       enabled;        /* flag: device has all resources allocated */
    _Bool       server;         /* driver acts as a server */
    NkDevVlink* vlink;          /* vlink */
    ExRing*     ring;           /* circular ring descriptor */
    size_t      size;           /* size of circular ring */
    size_t      pos;            /* reading/writing position inside ring */
    NkXIrq      s_xirq;         /* server side xirq */
    NkXIrq      c_xirq;         /* client side xirq */
    NkXIrqId    xid;            /* cross interrupt handler id */
    MUTEX       lock;           /* mutual exclusion lock for all ops */
    WAIT_QUEUE  wait;           /* waiting queue for all ops */
    int         count;          /* usage counter */
```

```
    /* Statistics */
    unsigned     opens;
    unsigned     reads;
    unsigned     writes;
    unsigned long long read_bytes;
    unsigned long long written_bytes;
    char     name[16];
} ExDev;
```

The *enabled* field is set to 1 when the *ExDev* is correctly allocated and all fields have been successfully initialized. The *server* field is set when the driver acts as a backend driver. In other words when the **vlink->s_id** is equal to the current VM identifier returned by the NKDDI nk_id_get primitive.

The fields **s_xirq** and **c_xirq** hold respectively the virtual interrupt number for the server cross interrupt, and the virtual interrupt number for the client cross interrupt (see nk_pxirq_alloc manual page for further details).

The frontend driver sends the **s_xirq** virtual interrupt to the backend driver to inform it that the circular buffer became non empty. In its turn the backend driver sends the **c_xirq** virtual interrupt to the frontend driver to inform it that the circular buffer became non-full.

The vpipe device driver registers its devices as regular character devices, so a user application can use standard system calls as open, close, read, write and select. Note that lseek system call is not allowed for an obvious reasons.

The vpipe device driver exports the following basic operation to the generic character device framework available in the Linux kernel: **ex_open**, **ex_release** **ex_read**, **ex_write** and **ex_poll**. All of them (except **ex_poll**) use a mutual exclusion mechanism to ensure that only one thread is performing a service at time.

A single waiting queue is implemented for all blocking operations (**wait**). All virtual interrupt handlers for cross interrupts (**ex_xirq_hdl** and *NK_XIRQ_SYSCONF* interrupts (**ex_sysconf_hdl**, see nk_xirq_trigger manual page for further details) always wake up all pending threads to execute handler processing. The sleeping primitive is implemented through the Linux kernel primitive **wait_event**, so awaken threads will recheck its sleeping conditions and perform appropriate actions. In addition, all sleeping conditions always check the peer driver status. If this status is not set to *NK_DEV_VLINK_ON* state, the current pending operation is aborted.

The *count* field is used to keep track of multiple **ex_open** operations on the same vpipe.

1.2.21.8 SEE ALSO

nk_pmem_alloc

nk_mem_map

nk_vtop

nk_ptov

nk_vlink_lookup

nk_xirq_trigger

nk_pxirq_alloc

nk_xirq_attach

nk_xirq_detach

1.2.22 VRPC(4D)

1.2.22.1 NAME

vrpc — Virtual RPC driver interface for both back-end and front-end sides.

1.2.22.2 SYNOPSIS

#include <nk/nkern.h>
#include <vlx/vrpc_common.h>
#include <vrpc.h>

The virtual RPC driver implements a generic Remote Procedure Call protocol between a backend and a frontend driver.

1.2.22.3 FEATURES

The virtual RPC should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual RPC inter-VM communication module

Virtual RPC devices must be declared in the platform device tree. The example below declares vclk devices using the virtual link framework(see the vlink_lookup manual page for more details).

```
&vm2_vdevs {

    vclk_ctrl_be: vclk_ctrl@be {         // vCLK control back-end for VM2
        compatible = "vrpc";             // uses generic vRPC protocol
        #clone     = <2>;                // 2 server end points
        server;                          //
        info       = "vclk_ctrl";        // device name
   };

};

&vm3_vdevs {

    vclk_ctrl@fe {                       // vCLK control front-end for VM3
        peer-phandle = <&vclk_ctrl_be>;  // peer vLINK
        client;                          //
        info         = "vclk_ctrl";      // device name
    };
};

&vm4_vdevs {

    vclk_ctrl@fe {                       // vCLK control front-end for VM4
        peer-phandle = <&vclk_ctrl_be>;  // peer vLINK
        client;                          //
        info         = "vclk_ctrl";      // device name
    };
};
```

1.2.22.4 DESCRIPTION

The virtual RPC driver offers an API in order to manage a remote procedure call from one end of a virtual link (the client side), to the other end of the virtual link (the server side).

1.2.22.5 EXTENDED DESCRIPTION

The virtual RPC API allows to:

- lookup virtual RPC devices;
- get various information from a virtual RPC device;
- open and close a virtual RPC device;
- do the remote procedure call.

1.2.22.6 LOOKING UP A VIRTUAL RPC DEVICE

1.2.22.6.1 SYNOPSIS

Before using a virtual RPC device, a backend driver must find it. This must be done by invoking the **vrpc_server↩ _lookup** function.

#include <vrpc.h>

struct vrpc_t∗ **vrpc_server_lookup** (char∗ *name*, struct vrpc_t∗ *last*);

1.2.22.6.2 DESCRIPTION

The above function looks up and locks the server endpoint of a virtual RPC device named *name*.

The *last* parameters specifies the start of the lookup process. If it is null, the lookup starts from the beginning of the virtual RPC device list.

This routines returns an handle on the found virtual RPC device or a null pointer if none was found.

1.2.22.6.3 SYNOPSIS

An endpoint of a virtual RPC device can be unlocked by invoking the **vrpc_release** function.

#include <vrpc.h>

void **vrpc_release** (struct vrpc_t∗ *vrpc*);

1.2.22.6.4 DESCRIPTION

The above function unlocks an endpoint of a virtual RPC device.

The *vrpc* parameter must be a valid virtual RPC device pointer previously obtained with a server or a client lookup.

1.2.22.7 INFORMATION FROM A VIRTUAL RPC DEVICE

1.2.22.7.1 SYNOPSIS

#include <vrpc.h>

NkOsId **vrpc_peer_id** (struct vrpc_t∗ *vrpc*);

1.2.22.7.2 DESCRIPTION

This routine returns the identifier of the VM using the other endpoint of the virtual RPC device.

1.2.22.7.3 SYNOPSIS

#include <vrpc.h>

void∗ **vrpc_data** (struct vrpc_t∗ *vrpc*);

1.2.22.7.4 DESCRIPTION

This routine returns the address of the RPC data buffer.

1.2.22.7.5 SYNOPSIS

#include <vrpc.h>

vrpc_size_t **vrpc_maxsize** (struct vrpc_t∗ *vrpc*);

1.2.22.7.6 DESCRIPTION

This routine returns the size of the RPC data buffer.

1.2.22.8 OPENING AND CLOSING A VIRTUAL RPC DEVICE

1.2.22.8.1 SYNOPSIS

#include <vrpc.h>

int **vrpc_server_open** (struct vrpc_t∗ *vrpc*, vrpc_call_t *call*, void∗ *cookie*, int *direct*);

1.2.22.8.2 DESCRIPTION

This routine can be used by a backend driver in order to specify the routine that will be used to implement the RPC call.

The first parameter *vrpc* is a pointer to a valid RPC device.

The second parameter *call* is the RPC routine and *cookie* the argument passed to the routine.

The last parameter specifies if the call can be made directly from the cross interrupt routine or from a thread context.

1.2.22.8.3 RETURN VALUES

This routine returns 0 in case of success or *-EFAULT* in case of failure.

1.2.22.8.4 SYNOPSIS

#include <vrpc.h>

int **vrpc_client_open** (struct vrpc_t∗ *vrpc*, vrpc_ready_t *ready*, void∗ *cookie*);

1.2.22.8.5 DESCRIPTION

This routine must be used by a frontend RPC client to connect to a backend RPC server.

The first parameter *vrpc* is a pointer to a valid RPC device.

The second parameter *ready* is a callback routine that will be invoked when the server side is ready, and *cookie* the argument passed to the routine.

1.2.22.8.6 RETURN VALUE

This routine returns 0 in case of success or *-EFAULT* in case of failure.

1.2.22.8.7 SYNOPSIS

#include <vrpc.h>

void **vrpc_close** (struct vrpc_t∗ *vrpc*);

1.2.22.8.8 DESCRIPTION

This routine can be used by a backend or a frontend RPC driver to close the RPC connection.

1.2.22.9 THE RPC CALL

1.2.22.9.1 SYNOPSIS

#include <vrpc.h>

int **vrpc_call** (struct vrpc_t∗ *vrpc*, vrpc_size_t∗ *size*);

1.2.22.9.2 DESCRIPTION

This routine can be used by a frontend RPC client to do an RPC call to the backend server.

The first parameter *vrpc* is a pointer to a valid RPC device.

The second parameter *size* contains the size of the RPC request. After the call it is updated to the size of the RPC response.

1.2.22.9.3 RETURN VALUES

This routine returns 0 in case of success, *-EAGAIN* if the RPC has been aborted and *-EFAULT* otherwise.

1.2.22.9.4 SYNOPSIS

#include <vrpc.h>

int **vrpc_call_busy** (struct vrpc_t∗ *vrpc*, vrpc_size_t∗ *size*, nku32_f *timeout_ms*);

1.2.22.9.5 DESCRIPTION

This routine can be used by a frontend RPC client to do an RPC call to the backend server, and busy-wait for its completion.

The first parameter *vrpc* is a pointer to a valid RPC device.

The second parameter *size* contains the size of the RPC request. After the call it is updated to the size of the RPC response.

The third parameter *timeout_ms* is the maximum duration to wait for the completion of the call, in milliseconds. If the call doesn't complete within this duration, it aborts and returns *-ETIMEDOUT*. Just because a request has errored with *-ETIMEDOUT* does not mean it did not, or will not complete in the server. The current implementation serializes requests, and the server will only respond to another request after completing the one which appears to have timed out.

1.2.22.9.6 RETURN VALUES

This routine returns 0 in case of success, *-ETIMEDOUT* if the RPC timed out, *-EAGAIN* if the RPC has been aborted for other reasons, and *-EFAULT* otherwise.

1.2.22.10 SEE ALSO

nk_pmem_alloc

nk_mem_map

nk_pxirq_alloc

nk_ptov

vlink_lookup

nk_vtop

nk_xirq_attach

nk_xirq_detach

nk_xirq_trigger

1.2.23 VRPQ_BE(4D)

1.2.23.1 NAME

vrpq_be — virtual Remote Procedure Queue back-end

1.2.23.2 DESCRIPTION

The vRPQ service provides a way for Linux user space applications to send batches of asynchronous requests from a client to a server. The client and server may run on different Virtual Machines. Requests are transmitted within "sessions" created on top of "channels". A server may simultaneously accept requests from different Virtual Machines. Several clients may also run on the same Virtual Machine. A "channel" is usually created from each Virtual Machine running a client. Clients running on a given Virtual Machine use different sessions.

The service is available from user space applications via a user space library which transparently invokes the services provided by the underlying Linux kernel module.

The service is provided by a Linux kernel module. There are two drivers: one front-end driver and one back-end driver. The service relies on the Hypervisor vlink service.

1.2.23.3 CONFIGURATION

1.2.23.3.1 Linux Kernel Configuration

The vRPQ driver is usually enabled as a dependency of another virtual driver relying on the vRPQ services. For example, below is an excerpt of the Linux Kconfig file describing the vOpenGL driver:∗

```
config VOGL_BE
    tristate "Virtual OpenGL ES back-end infrastructure"
    depends on m
    default n
    select VRPQ_BE
    ...
```

The vRPQ back-end driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> vRPQ_BE

It can be compiled as a loadable module or as an embedded driver.

1.2.23.3.2 Device Tree Configuration

There is no specific directive in the platform device tree to enable this module. The size of the persistent global memory used for message queuing is given with a virtual back-end driver relying on this library, such as the v↩ OpenGL one used below.

The example below shows a Device Tree configuration including both the back-end node and the front-end node.

The vRPQ service here is used by the vOpenGL feature.

```
&vm2_vdevs {
    vogl_be: vogl@be {          // vOpenGL back-end for VM3
           compatible = "vogl";
           info       = "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
   };
};
```

1.2.23.4 USER SPACE DESCRIPTION

1.2.23.4.1 SYNOPSIS

#include <vlx/vrpq/vrpq.h> #include "lib/vrpq.h"

The VRPQ API for client library is briefly described in the **vlx/vrpq/client-lib.h** file.

vRPQ queues on the server side appear as character devices within the Linux file system. The configuration presented above would lead to the creation of a /dev/vrpq-srv-vogl0 device on the server (back-end) side. It is up to the virtual driver using the vRPQ back-end driver to trigger the creation of such a character device in the file system.

1.2.23.5 /proc/nk Entries

There is no vRPQ information available through the /proc/nk directory.

1.2.23.6 KERNEL DESCRIPTION

1.2.23.6.1 SYNOPSIS

#include <vrpq/vrpq.h>

The virtual Remote Procedure Queue (VRPQ) back-end (server) library module is integrated in Linux and provides an interface to virtual back-end drivers for managing message queuing.

The VRPQ API for server driver is briefly described in **vlx/vrpq/server-drv.h** file.

1.2.23.6.2 Data Structure

The VRPQ back-end library provides an interface to manage message queuing for virtual back-end drivers. This interface is mainly used for vOpenGL on top of the Hypervisor. The VRPQ back-end library is based on the virtual Link Wrapper Library (see the vlink_lib manual page for a full description of the interface exported by this library).

The VRPQ back-end library provides three main primitives for virtual back-end drivers described in the following sections. This library exports a *VrpqDrv* object whose fields are briefly described below:

```
typedef int (*VrpqPropGet) (Vlink* vlink, unsigned int type, void* prop);

typedef struct VrpqDrv {
    /* Public */
    const char*              name;
    unsigned int             major;
    unsigned int             resc_id_base;
    VrpqPropGet              prop_get;
    VrpqParentContext*       parent_context;
    /* Internal */
    unsigned int             major_reg;
    VlinkDrv*                parent_drv;
    VlinkOpDesc*             vops;
    VrpqDevOps               dops;
    struct VrpqIFDrv*        if_drv;
    struct VrpqDev*          devs;
    struct list_head         link;
} VrpqDrv;
```

The public fields are filled by the caller as follows:

name

A string of characters containing the name of the virtual device in /**dev** directory

major

The required major number for the virtual device. If this field has value 0, a major will be allocated dynamically by the Linux kernel. The actually used value will be stored in the *major_reg* field

resc_id_base

The resource identifier base number. This value is used by the VRPQ back-end library for allocating persistent resources through nk_pdev_alloc and nk_pmem_alloc primitives

prop_get

A pointer to a valid callback routine returning 0 and setting the size of the property in the third argument if the type of the property related to the virtual back-end driver is found. If the property is not found, the size of the property is not updated. If the type of property is incorrect, *-EINVAL* is returned. This field cannot be a null pointer in this version

parent_context

A pointer to the parent context

All the internal fields are filled by the VRPQ back-end library (see section VRPQ SERVER INITIALIZATION for further details).

1.2.23.6.3 VRPQ SERVER INITIALIZATION

1.2.23.6.3.1 SYNOPSIS

This primitive is invoked to initialize VRPQ on the server side (back-end side).

#include <vrpq/vrpq.h>

int **vrpq_srv_drv_init** (VlinkDrv∗ *parent_drv*, VrpqDrv∗ *vrpq_drv*);

1.2.23.6.3.2 DESCRIPTION

This primitive initializes on the back-end side a *VrpqDrv* object. This primitive is invoked by the virtual link wrapper library when a virtual back-end driver has successfully probed its associated virtual device through the **int vlink_drv_probe(VlinkDrv∗)** virtual link wrapper library (see the vlink_lib manual page for further details).

1.2.23.6.3.3 PARAMETERS

The first **parent_drv** argument is a valid pointer to a *VlinkDrv* object whose fields are described in the vlink_lib manual page. The caller provides three callback routines to initialize and cleanup virtual links and to initialize the virtual back-end driver.

The second **vrpq_drv** argument is a valid pointer to a *VrpqDrv* object whose public fields (see section DESCRIP←TION for further details) have been previously updated.

In case of success, the private fields of the second arguments are updated with the appropriate information.

The **parent_drv** field is filled with the first argument given to this primitive.

The **vops** field is filled with a set of internal virtual User Memory Buffer described in the following sections back-end library for virtual link management:

int vrpq_srv_vlink_reset(Vlink*, void*)

To reset a virtual link related to a VRPQ on the server side

int vrpq_srv_vlink_start(Vlink*, void*)

To start up communication on virtual links related to a VRPQ on the server side

int vrpq_srv_vlink_abort(Vlink*, void*)

To abort virtual links related to a VRPQ on the server side

int vrpq_srv_vlink_stop(Vlink*, void*)

To stop communication through virtual links related to a VRPQ on the server side

int vrpq_srv_vlink_cleanup(Vlink*, void*)

To remove resources used for virtual links related to a VRPQ on the server side

These callback routines are also described in the vlink_lib manual page.

The **fops** field is updated with a table of functions invoked to manage the /**dev**/**XXXX** virtual device:

int vrpq_srv_open(struct inode*, struct file*)

To open the associated virtual device on the server side

int vrpq_srv_release(struct inode*, struct file*)

To release the associated virtual device on the server side

int vrpq_srv_ioctl(struct file*, unsigned int, unsigned long)

To perform a specific operation on the associated virtual device on the server side

All other primitives related to this virtual device are not implemented in this version&.

1.2.23.6.3.4 RETURN VALUES

This primitive returns 0 in case of success, otherwise the following error codes are returned:

-ENOMEM

Not enough resource to perform this primitive

-EBUSY

Another virtual back-end driver for user memory management is already connected to the **major** device number

-EINVAL or any positive error code

If parameters are incorrect or if the driver cannot be registered in the Linux kernel.

1.2.23.6.4 VRPQ SERVER CLEANUP

1.2.23.6.4.1 SYNOPSIS

This primitive is called by a virtual back-end driver to delete a previous initialization done on the server side, through the primitive described in section VRPQ SERVER INITIALIZATION.

#include <vrpq/vrpq.h>

void **vrpq_srv_drv_cleanup** (VrpqDrv∗ *vrpq_drv*);

1.2.23.6.4.2 DESCRIPTION

This primitive releases and performs the associated cleanup when a *VrpqDrv* object is released by a virtual back-end driver.

1.2.23.6.5 PARAMETERS

The unique **vrpq_drv** argument is a valid pointer to a *VrpqDrv* object which has been previously updated by the primitive described in VRPQ SERVER INITIALIZATION section.

1.2.23.6.5.1 RETURN VALUES

No relevant value is returned by this primitive.

1.2.23.6.6 VRPQ SERVER VIRTUAL LINK INITIALIZATION

1.2.23.6.6.1 SYNOPSIS

This primitive is invoked by the virtual link wrapper library to initialize all virtual links related to the VRPQ on the server side (back-end side).

#include <vrpq/vrpq.h>

int **vrpq_srv_vlink_init** (VrpqDrv∗ *vrpq_drv*, struct Vlink∗ *vlink*);

1.2.23.6.6.2 DESCRIPTION

This primitive is invoked by the virtual link wrapper library (from **vlink_drv_startup(VlinkDrv∗)**) after having successfully initialized the associated virtual back-end driver (see the vlink_lib manual page for further details).

1.2.23.6.6.3 PARAMETERS

The first **vrpq_drv** argument is a valid pointer to a *VrpqDrv* object which has been previously initialized by **vrpq_srv_drv_init(VlinkDrv∗, VrpqDrv∗)** primitive.

The second **vlink** argument is a valid pointer to a *struct Vlink* object whose fields are described in the **vlink_lib(4D)** manual page.

This primitive gets the property of the associated virtual back-end driver by invoking the callback routine previously hooked by the associated virtual back-end driver in **prop_get** field of the *VrpqDrv* object (see section DESCRIPTION for further details).

1.2.23.6.7 RETURN VALUES

In case of success, this primitive returns 0, otherwise a non null error code is returned.

1.2.23.7 SEE ALSO

nk_pmem_alloc

nk_pxirq_alloc

vrpq_fe

vlink_lib

1.2.24 VRPQ_FE(4D)

1.2.24.1 NAME

vrpq_fe — virtual Remote Procedure Queue front-end

1.2.24.2 DESCRIPTION

The vRPQ service provides a way for Linux user space applications to send batches of asynchronous requests from a client to a server. The client and server may run on different Virtual Machines. Requests are transmitted within "sessions" created on top of "channels". A server may simultaneously accept requests from different Virtual Machines. Several clients may also run on the same Virtual Machine. A "channel" is usually created from each Virtual Machine running a client. Clients running on a given Virtual Machine use different sessions.

The service is available from user space applications via a user space library which transparently invokes the services provided by the underlying Linux kernel module.

The service is provided by a Linux kernel module. There are two drivers: one front-end driver and one back-end driver. The service relies on the Hypervisor vlink service.

1.2.24.3 CONFIGURATION

1.2.24.3.1 Linux Kernel Configuration

The vRPQ driver is usually enabled as a dependency of another virtual driver relying on the vRPQ services. For example, below is an excerpt of the Linux Kconfig file describing the vOpenGL driver:∗

```
config VOGL_FE
        tristate "Virtual OpenGL ES front-end infrastructure"
        depends on m
        default n
        select VRPQ_FE
    ...
```

The vRPQ driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> vRPQ_FE

It can be compiled as a loadable module or as an embedded driver.

1.2.24.3.2 Device Tree Configuration

There is no specific directive in the platform device tree to enable this module. The size of the persistent global memory used for message queuing is given with a virtual front-end driver relying on this library, such as the v↩ OpenGL one used below.

The example below shows a Device Tree configuration including both the back-end node and the front-end node.

The vRPQ service here is used by the vOpenGL feature.

```
&vm2_vdevs {
    vogl_be: vogl@be {           // vOpenGL back-end for VM3
           compatible = "vogl";
           info       = "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
   };
};
```

1.2.24.4 USER SPACE DESCRIPTION

1.2.24.4.1 SYNOPSIS

#include <vlx/vrpq/vrpq.h> #include "lib/vrpq.h"

The VRPQ API for client library is briefly described in the **vlx/vrpq/client-lib.h** file.

vRPQ queues on the client side appear as character devices within the Linux file system. The configuration presented above would lead to the creation of a /dev/vrpq-clt-vogl0 device on the client (front-end) side. It is up to the virtual driver using the vRPQ front-end driver to trigger the creation of such a character device in the file system.

1.2.24.5 /proc/nk Entries

There is no vRPQ information available through the /proc/nk directory.

1.2.24.6 KERNEL DESCRIPTION

1.2.24.6.1 SYNOPSIS

#include <vrpq/vrpq.h>

The virtual Remote Procedure Queue (VRPQ) front-end (client) library module is integrated in Linux and provides an interface to virtual front-end drivers for managing message queuing.

The VRPQ API for client driver is briefly described in **vlx/vrpq/client-drv.h** file.

The VRPQ front-end library provides an interface to manage message queuing for virtual front-end drivers. This interface is mainly used for vOpenGL on top of the Hypervisor. The VRPQ front-end library is based on the virtual Link Wrapper Library (see the vlink_lib manual page for a full description of the interface exported by this library).

The VRPQ front-end library provides three main primitives for virtual front-end drivers, which are described in the following sections. This library exports a *VrpqDrv* object whose fields are explained in the vrpq_be manual page.

The VRPQ front-end library provides three main similar primitives for virtual front-end drivers described in the following sections.

The public fields of the *VrpqDrv* object are filled with similar information as the virtual back-end driver (see the vrpq_be manual page for further details).

1.2.24.6.2 VRPQ CLIENT INITIALIZATION

1.2.24.6.2.1 SYNOPSIS

This primitive is invoked to initialize VRPQ on the client side (front-end side).

#include <vrpq/vrpq.h>

int **vrpq_clt_drv_init** (VlinkDrv∗ *parent_drv*, VrpqDrv∗ *vrpq_drv*);

1.2.24.6.2.2 DESCRIPTION

This primitive initializes on the front-end side a *VrpqDrv* object. It is invoked by the virtual link wrapper library when a virtual front-end driver has successfully probed its associated virtual device through the **int vlink_drv_probe(↩ VlinkDrv∗)** virtual link wrapper library (see the vlink_lib manual page for further details).

1.2.24.6.2.3 PARAMETERS

The first **parent_drv** argument is a valid pointer to a *VlinkDrv* object whose fields are described in the vlink_lib manual page. The caller provides three callback routines to initialize and cleanup virtual links and to initialize the virtual front-end driver.

The second **vrpq_drv** argument is a valid pointer to a *VrpqDrv* object whose public fields (see the vrpq_be manual page for further details) has been previously updated.

In case of success, the private fields of the second argument are updated with the appropriate information.

The **parent_drv** field is filled with the first argument given to this primitive.

The **vops** field is filled with a set of internal VRPQ front-end library for virtual link management:

int vrpq_clt_vlink_reset(Vlink*, void*)

To reset a virtual link related to a VRPQ on the client side

int vrpq_clt_vlink_start(Vlink*, void*)

To start up communication on virtual links related to a VRPQ on the client side

int vrpq_clt_vlink_abort(Vlink*, void*)

To abort virtual links related to a VRPQ on the client side

int vrpq_clt_vlink_stop(Vlink*, void*)

To stop communication through virtual links related to a VRPQ on the client side

int vrpq_clt_vlink_cleanup(Vlink*, void*)

To remove resources used for virtual links related to a VRPQ on the client side

These callback routines are also described in the vlink_lib manual page.

The **dops** field is updated with a table of functions invoked to manage the /**dev**/**XXXX** virtual device:

int vrpq_clt_open(struct inode*, struct file*)

To open the associated virtual device on the client side

int vrpq_clt_release(struct inode*, struct file*)

To release the associated virtual device on the client side

int vrpq_clt_ioctl(struct file*, unsigned int, unsigned long)

To perform a specific operation on the associated virtual device on the client side. A dedicated operation is devoted to profiling and can be enabled in the Linux configuration file. This specific operation is described in section VRPQ PROFILING.

All other primitives related to this virtual device are not implemented in this version.

1.2.24.6.2.4 RETURN VALUES

This primitive returns 0 in case of success, otherwise the following error codes are returned:

-ENOMEM

Not enough resource to perform this primitive

-EBUSY

Another virtual front-end driver for user memory management is already connected to the **major** device number

-EINVAL or any positive error code

If the driver cannot be registered in the Linux kernel

1.2.24.6.3 VRPQ PROFILING

This dedicated operation can be invoked to the get statistics about VRPQ. A list of available parameters is briefly described in **vlx/vrpq/common.h** file (*VRPQ_PROF_XXX*). Statistics are also available per procedure (function and group, see the VRPQ API client library described in **vlx/vrpq/client-lib.h** file for further details). The following commands are available:

VRPQ_PROF_GEN_STAT_GET

Get a parameter value

VRPQ_PROF_STAT_GET

Get statistics related to a procedure

VRPQ_PROF_STAT_GET_ALL

Get full statistics: for all parameters and procedures

VRPQ_PROF_RESET

Reset all statistics

1.2.24.6.4 VRPQ CLIENT CLEANUP

1.2.24.6.4.1 SYNOPSIS

This primitive is called by a virtual front-end driver to delete a previous initialization done on the server side through the primitive described in section VRPQ CLIENT INITIALIZATION.

#include <vrpq/vrpq.h>

void **vrpq_clt_drv_cleanup** (VrpqDrv∗ *vrpq_drv*);

1.2.24.6.4.2 DESCRIPTION

This primitive releases and performs the associated cleanup when a *VrpqDrv* object is released by a virtual front-end driver.

1.2.24.6.4.3 PARAMETERS

The unique *vrpq_drv* argument is a valid pointer to a *VrpqDrv* object which has been previously updated by the primitive described in VRPQ CLIENT INITIALIZATION section.

1.2.24.6.4.4 RETURN VALUES

No relevant value is returned by this primitive.

1.2.24.6.5 VRPQ CLIENT VIRTUAL LINK INITIALIZATION

1.2.24.6.5.1 SYNOPSIS

This primitive is invoked by the virtual link wrapper library to initialize all virtual links related to the VRPQ on the client side (front-end side).

#include <vrpq/vrpq.h>

int **vrpq_clt_vlink_init** (VrpqDrv∗ *vrpq_drv*, struct Vlink∗ *vlink*);

1.2.24.6.5.2 DESCRIPTION

This primitive is invoked by the virtual link wrapper library (from **vlink_drv_startup(VlinkDrv∗)**) after successfully initializing the associated virtual front-end driver (see the vlink_lib manual page for further details).

1.2.24.6.5.3 PARAMETERS

The first **vrpq_drv** argument is a valid pointer to a *VrpqDrv* object which has been previously initialized by **vrpq_clt_drv_init(VlinkDrv∗, VrpqDrv∗)** primitive.

The second **vlink** argument is a valid pointer to a *struct Vlink* object whose fields are described in the vlink_lib manual page.

This primitive gets the property of the associated virtual front-end driver by invoking the callback routine previously hooked by the associated virtual front-end driver in **prop_get** field of the *VrpqDrv* object (see DESCRIPTION for further details).

1.2.24.6.5.4 RETURN VALUES

In case of success, this primitive returns 0, otherwise a non null error code is returned.

1.2.24.7 SEE ALSO

vrpq_be

vlink_lib

1.2.25 VRTC_BE(4D)

1.2.25.1 NAME

vrtc_be — Virtual Real Time Clock back-end driver

1.2.25.2 SYNOPSIS

#include <nk/nkern.h>
#include <vlx/vrtc_common.h>
#include <vrpc.h>

The virtual real time clock (vRTC) backend driver is based on virtual RPC (see vrpc manual page for further details) provided by the Hypervisor and runs on top of Linux. In this current version, the backend driver relies upon a real RTC (Real Time Clock) device and defined in the **include/linux/rtc.h** file.

1.2.25.3 FEATURES

The virtual RTC frontend driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual Real Time Clock backend interface

It can be compiled as a loadable module or as an embedded driver.

Virtual RTC devices must be declared in the platform device tree. The example below declares vrtc devices using the virtual link framework(see the nk_vlink_lookup manual page for more details).

```
&vm2_vdevs {

    vrtc_be: vrtc@be {               // vRTC backend for VM3 and VM4
        compatible = "vrpc";         // uses generic vRPC protocol
        #clone     = <2>;            // 2 server end points
        server;                      //
        info       = "vrtc";         // device name
   };

};

&vm3_vdevs {

    vrtc@fe {                        // VM3 vRTC front-end
        peer-phandle = <&vrtc_be>;   // peer vLINK
        client;                      // frontend end point
        info         = "vrtc";       // device name
    };
};

&vm4_vdevs {

    vrtc@fe {                        // VM4 vRTC front-end
        peer-phandle = <&vrtc_be>;   // peer vLINK
        client;                      // frontend end point
        info         = "vrtc";       // device name
    };
};
```

There are two virtual links in this example. The server side is managed by the backend vrtc driver running on VM #2. A client side is managed by a frontend vrtc driver running on VM #3 and a second client side is managed by a frontend vrtc driver running on VM #4.

The 'info' property is only used for giving the device name.

1.2.25.4 DESCRIPTION

The vRTC backend driver provides an API to frontend drivers running on other VMs. The backend driver is responsible for accessing the physical device through an underlying native driver. This API is also accessible by frontend drivers running on other VM's. Multiple vRTC frontend drivers can be connected to a single peer vRTC backend driver.

1.2.25.5 EXTENDED DESCRIPTION

The vRTC backend driver is an abstraction which enables a vRTC frontend driver to access a RTC device managed by the backend driver. The vRTC driver uses a vrpc link to provide communications and synchronization between frontend and backend drivers.

At initialization time, the backend driver tries to associate each physical RTC device with virtual links present in the platform device tree. The vrpc primitives **vrpc_peer_id()** and **vrpc_server_lookup()** are used for this purpose. In the initialization module routine **vrtc_init()**, a delayed approach is carried out in order to wait until the frontend is booted (see **vrtc_setup()** for further details). If no RTC is detected, (that is, **rtc_class_open()** fails, the vRTC backend driver cannot be initialized and the module is not loaded (returning *-ESRCH*).

Then, the vRTC backend driver creates and initializes virtual RTC devices using **vrpc_data** and **vrpc_maxsize** and creates each virtual RTC through **vrtc_create()**. This latter primitive is based upon vrpc services and NKDDI services such as **nk_pmem_alloc(3D)**, **nk_pxirq_alloc(3D)** and **nk_xirq_attach(3D)**. Finally, it opens the virtual RPC connection using **vrpc_server_open()**: the server is now ready to execute the RPC requests coming from any sites through **vrtc_process_calls()**. The next section explains in detail all the services exported by the backend driver.

1.2.25.6 VIRTUAL REAL TIME CLOCK RPC PROTOCOL

The vRTC backend and frontend drivers exchange data through virtual links. Frontend requests are processed by the backend driver and a result is sent back to the frontend. The objects *vrtc_req_t* for requests and *vrtc_res_t* for backend responses have the following layout:

```
    /* VRTC RPC request */
typedef struct {
    nku32_f vcmd;        /* vrtc_cmd_t */
    nku32_f arg;         /* argument (optional) */
} vrtc_req_t;

    /* VRTC RPC result */
typedef struct {
    nku32_f res;         /* result */
    nku32_f value;       /* value */
} vrtc_res_t;
```

Each vRTC frontend driver send requests to the vRTC backend driver and a reply is given back. When a service has been successfully executed, the **res** field of *vrtc_res_t* object is set to 0, otherwise a negative error code is returned.

Each request is filled with a valid request held in **vcmd** field of a *vrtc_req_t* object. Additional data may be also updated in **arg** field of this object according to the request (ioctl services, set state and set frequency). The following possible services exported by the vRTC backend driver are:

```
typedef enum {
    VRTC_CMD_OPEN,                /* Open the RTC device */
    VRTC_CMD_RELEASE,             /* Release the RTC device */
    VRTC_CMD_IOCTL_RTC_AIE_ON,    /* Alarm Interrupt On */
    VRTC_CMD_IOCTL_RTC_AIE_OFF,   /* Alarm Interrupt Off */
    VRTC_CMD_IOCTL_RTC_UIE_ON,    /* Update Interrupt On */
    VRTC_CMD_IOCTL_RTC_UIE_OFF,   /* Update Interrupt Off */
    VRTC_CMD_IOCTL_RTC_PIE_ON,    /* Periodic Interrupt On */
    VRTC_CMD_IOCTL_RTC_PIE_OFF,   /* Periodic Interrupt Off */
    VRTC_CMD_IOCTL_RTC_WIE_ON,    /* Watchdog Interrupt On */
    VRTC_CMD_IOCTL_RTC_WIE_OFF,   /* Watchdog Interrupt Off */
    VRTC_CMD_READ_TIME,           /* Read current time (date) */
    VRTC_CMD_SET_TIME,            /* Set current time (date) */
    VRTC_CMD_READ_ALARM,          /* Read current alarm (date) */
    VRTC_CMD_SET_ALARM,           /* Set current alarm (date) */
    VRTC_CMD_IRQ_SET_STATE,       /* Enable/Disable 2^n periodic IRQs */
    VRTC_CMD_IRQ_SET_FREQ,        /* Set @^n Hz periodic IRQ frequency */
    VRTC_CMD_MAX                  /* Last slot, not a command */
} vrtc_cmd_t;
```

The backend vRTC driver decodes, executes and sets the response using the same virtual RPC shared memory region. Then, the frontend driver is unblocked from its **vrpc_call()**.

For ioctl commands, if the physical RTC driver has no ioctl, the vRTC backend driver returns *-ENOIOCLCMD* error code, otherwise the returned code is set to the returned value of the physical RTC driver. For open and release, 0 is always returned to the frontend driver.

For read/set time and read/set alarm a specific data structure is returned as an union of various objects:

```
typedef union {
    vrtc_req_t          req;
    vrtc_req_time_t     req_time;
    vrtc_req_wkalrm_t   req_wkalrm;
    vrtc_res_t          res;
    vrtc_res_time_t     res_time;
    vrtc_res_wkalrm_t   res_wkalrm;
} vrtc_ipc_t;
```

The requests for time have the following layout:

```
typedef struct {
    vrtc_req_t    common;
    vrtc_time_t   time;
} vrtc_req_time_t;
```

A time/Alarm read/set request is recorded as full date and has the following layout:

```
typedef struct {
    nku32_f     tm_sec;
    nku32_f     tm_min;
    nku32_f     tm_hour;
    nku32_f     tm_mday;
    nku32_f     tm_mon;
    nku32_f     tm_year;
    nku32_f     tm_wday;
    nku32_f     tm_yday;
    nku32_f     tm_isdst;
} vrtc_time_t;
```

The requests to read/set from/to alarm have the following layout:

```
typedef struct {
    vrtc_req_t      common;
    vrtc_wkalrm_t   alarm;
} vrtc_req_wkalrm_t;
```

An alarm read/set request is recorded as the following object:

```
typedef struct {
    nku16_f     enabled;        /* Alarm enabled (1) or disabled (0) */
    nku16_f     pending;        /* Alarm pending (1) or not not pending (0) */
    vrtc_time_t time;           /* time the alarm is set to */
} vrtc_wkalrm_t;
```

A response sent back to a vRTC frontend driver for time read has the following layout:

```
typedef struct {
    vrtc_res_t  common;
    vrtc_time_t time;
} vrtc_res_time_t;
```

A response sent back to a vRTC front—end driver for alarm read has the following layout:

```
typedef struct {
    vrtc_res_t        common;
    vrtc_wkalrm_t     alarm;
} vrtc_res_wkalrm_t;
```

In addition, a specific interrupt handler is used to get alarm or watchdog interrupts: **vrtc_rtc_task(void∗ cookie)**. These interrupts belongs to the following events:

VRTC_EVENT_UF

For update events corresponding to the Linux kernel value *RTC_UF* related to update interrupt for 1Hz RTC

VRTC_EVENT_AF

For Alarm events corresponding to the Linux kernel value *RTC_AF* related to alarm interrupt

VRTC_EVENT_PF

For periodic events corresponding to the Linux kernel value *RTC_PF* related to periodic interrupt.

This handler is responsible to trigger the appropriate event to the frontend driver when an alarm, a periodic interrupt, or a watchdog interrupt is triggered on the current VM.

1.2.25.7 SEE ALSO

nk_id_get

nk_pmem_alloc

nk_pxirq_alloc

nk_xirq_attach

vlink_lookup

vrpc

vrtc_fe

1.2.26 VRTC_FE(4D)

1.2.26.1 NAME

vrtc_fe — Virtual Real Time Clock front-end driver

1.2.26.2 SYNOPSIS

#include <nk/nkern.h>
#include <vlx/vrtc_common.h>
#include <vrpc.h>

The virtual Real Time Clock (vRTC) frontend driver is based on virtual RPC (see vrpc for further details) provided by the Hypervisor and runs on top of Linux.

1.2.26.3 FEATURES

The virtual RTC frontend driver should be enabled in the Linux configuration file:

Device Drivers -> VLX virtual device support -> Virtual Real Time Clock frontend interface

It can be compiled as a loadable module or as an embedded driver.

Virtual devices are declared in the platform device tree for each VM. An example is given in the vrtc_be manual page.

1.2.26.4 DESCRIPTION

A vRTC frontend driver performs a virtual RPC to the backend to access a physical Real Time Clock device.

1.2.26.5 EXTENDED DESCRIPTION

At initialization time, the frontend driver tries to find the corresponding virtual RPC link. The vrpc primitives **vrpc_peer_id()** and **vrpc_client_lookup()** are used for this purpose.

Then, the driver creates and initializes the virtual RTC device using **vrpc_data** and **vrpc_maxsize** and opens the virtual RPC connection using **vrpc_client_open()** when the virtual RTC is probed in the **vrtc_probe()** function. This latter primitive is based upon vrpc services and NKDDI services such as nk_pmem_alloc, nk_pxirq_alloc and nk_xirq_attach. If the virtual RTC is correctly probed, the client is now ready to do virtual RPCs to the server.

1.2.26.6 VIRTUAL REAL TIME CLOCK RPC PROTOCOL

The frontend vRTC driver sets up the request using the virtual RPC shared memory region and triggers the call using **vrpc_call()**. Then it waits for the completion of the RPC. The **vrpc_call()** is encapsulated in **vrtc_call_ex()** function allowing to restart a remote procedure call when the link is down through a new invocation of **vrpc_client_open()** vRPC primitive.

All the data structures and protocol are described in the vrtc_be manual page.

The vRTC frontend driver can set into suspend mode for power management through the **vrtc_pm_suspend()** primitive, or woken up through the **vrtc_pm_resume()** primitive. When the frontend driver is suspended, **irq_wake_enabled** global counter is incremented, and decremented upon resume.

The vRTC frontend driver allows to read a **/proc/vrtc** file when the vRTC frontend driver is correctly registered. This file contains the following information:

VLX VRTC frontend. RTC is registered.

When the vRTC frontend driver has been correctly registered, internal vRTC fontend informations are appended in this file picked up from the *vrtc_t* global object:

RPC buffer 0xp cxirq d cxid 0xx IRQ pshared 0xlx

The RPC buffer address (*vrpc_data*), *cxirq* client crossinterrupt number, *cxid* client crossinterrupt identifier and the physical address of the memory chunk allocated from the global persitent memory pool (IRQ phared).

The next line has the following layout:

IRQ vshared 0xp IRQ events 0xx irq_wake_enabled d

The virtual address corresponds to *pshared*, the number of crossinterrupts received (IRQ events), and the current state of the *irq_wake_enabled* global counter (suspend 1, resume 0).

Then the statistics of calls is appended. The name of the call (open, release, aie_on, aie_off, uie_on, uie_off, pie_on, pie_off, wie_on, wie_off, read_time, set_time, read_alarm, set_alarm, irq_set_state, irq_set_freq) and the number of invocations done to the backend driver.

Finally, the total number of crossinterrupts received, and the total crossinterrupts number for Update, Alarms and Periodic events.

1.2.26.7 SEE ALSO

nk_id_get

nk_pmem_alloc

nk_pxirq_alloc

nk_xirq_attach

nk_vlink_lookup

vrpc

vrtc_be

1.2.27 VVIDEO2(4D)

1.2.27.1 Cross References

Related Documents
Manual Page

1.2.27.2 NAME

vvideo2 — Virtual Video v2

1.2.27.3 SYNOPSIS

The Virtual Video version 2 (vvideo2) backend and frontend drivers, when paired, enable userspace clients to access the functionality of Video For Linux v2 (V4L2) hardware devices (video capture, video overlays, video codecs and video scalers) from a virtual machine which does not host the physical hardware drivers. The vvideo2 backend driver receives requests from the frontend vvideo2 driver and forwards them to the underlying native video drivers via Linux kernel-mode APIs.

1.2.27.4 FEATURES

Specific V4L2 devices can be exported from backend to frontend guest, at the condition that they can operate using DMABUF memory buffers. Other memory buffer types are not supported.

vvideo2 also manages DMA fences associated with buffers, for *VIDIOC_QBUF* and *VIDIOC_DQBUF* ioctls. It uses services of the VFENCE2(4D) driver for that.

vvideo2 supports the *poll(2)* and *select(2)* system calls, as well as non-blocking operations, defined at *open(2)* time or with *fcntl(2)*.

Operations are zero-copy as far as video data itself are concerned, because the backend directly accesses frontend-allocated buffers containing compressed or uncompressed video data. Only protocol meta-data require copying between virtual machines.

When MFC (Multi Function Codec) and Scaler devices are exported on an ExynosAuto V9 platform, standard test and playback code can be executed in the frontend virtual machine unmodified, in parallel with execution in backend virtual machine.

Video processing in backend on behalf of frontend is separate from actual display (in case of decoding), which can happen in frontend virtual machine, using hardware which is controlled directly by that machine.

The vvideo2 backend driver optionally supports running in Google's Generic Kernel Image mode, where it is loaded dynamically and only uses a Google-approved subset of exported Linux kernel symbols. For this, it must be compiled with the `CONFIG_VLX_VVIDEO2_BE_VGKI` configuration option. It then uses the VGKI kernel-mode API instead of non-GKI calls, requires the presence of the VGKI(4D) driver to load and of the vgki-helper(8) process to perform requests. The vvideo2 frontend driver is GKI compatible by default.

Current state of vdrivers is visible in */proc/nk/vvideo2-fe* and */proc/nk/vvideo2-be* files.

```
console:/ # cat /proc/nk/vvideo2-fe
vVideo v2 frontend
links 1 dev_num 7 async_mask 0xf pending_releases 0 pending_link_ons 0
L.Mi Pr OC VI FsMi Path--------- ON FN A RB0 Name
 configs 7 msgs/used 32/1 link_on 1 session 1 pending_link_on 0
 Stats: link_on:1 link_off:1
0.0    2  0  0     0 /dev/video6    6 -1 S    0 s5p-mfc-dec0
 Stats: open:30 release:30 fd2gid:1064 gid2fd:1915 init3:256 init4:808 export0:808 export1:256 unexport0:1064
0.1    2  0  1     1 /dev/video7    7 -1 S    0 s5p-mfc-enc0
 Stats: open:10 release:10 fd2gid:76 gid2fd:144 init3:10 init4:66 export0:66 export1:10 unexport0:76 ioctl_err
0.2    2  0  2     2 /dev/video8    8 -1 S    0 s5p-mfc-dec-secure0
0.3    2  0  3     3 /dev/video9    9 -1 S    0 s5p-mfc-enc-secure0
0.4    2  0  4     4 /dev/video10  10 -1 S    0 s5p-mfc-enc-otf0
0.5    2  0  5     5 /dev/video11  11 -1 S    0 s5p-mfc-enc-otf-secure0
0.6    2  0  6     6 /dev/video50  50 -1 A 600 exynos5-scaler:m2m
 Stats: open:18 release:18 fd2gid:888 gid2fd:1776 init3:79 init4:809 export0:809 export1:79 unexport0:888 asyn
 Sl PID-- Name---- L.Mi ORAIFCK RF Transa Request- Result-- Ctx-- RC
  0      0 msg0      None O......  0       0 NONE             0      0  1
console:/ #

root@euto-v9-sadk:~# cat /proc/nk/vvideo2-be
vVideo v2 backend
links 1 sysconf 0 receive 0 max_reqs 32 threads/idle 1/1
```

```
 Stats: thread_starts:75 kills:46 stopped_kills:2 max_running:5
L.Mi Pr RC RC Config
 configs 7 dev_width/mask 4/0xf msgs 32 connected 1 receive 0
next_transaction 2: 10321 10322 10319 10320
0.0   3 .C  0 s5p-mfc-dec0
 Stats: fc3to2:513 gid2fd_alloc:1064
0.1   3 .C  0 s5p-mfc-enc0
 Stats: gid2fd_alloc:76
0.2   3 .C  0 s5p-mfc-dec-secure0
0.3   3 .C  0 s5p-mfc-enc-secure0
0.4   3 .C  0 s5p-mfc-enc-otf0
0.5   3 .C  0 s5p-mfc-enc-otf-secure0
0.6   3 .C  0 exynos5-scaler:m2m
 Stats: deferred_returns:3254 behind_async:474 next_none:2780 next_async:301 next_sync:173 gid2fd_alloc:888 fe
Sl MS R Transa Result-- Ctx-- AWD PID-- Name Current Device
 1 -1 0       0         0      0 ..0    616  th1 None    None
root@euto-v9-sadk:~#
```

1.2.27.5 PARAMETERS

The frontend driver should be enabled in the Linux configuration tree:

Device Drivers -> VLX virtual device support -> Virtual video v2 frontend interface

The backend driver should be enabled in the Linux configuration tree:

Device Drivers -> VLX virtual device support -> Virtual video v2 backend interface

Both drivers can be compiled as loadable/unloadable modules or as embedded drivers.

Virtual Video backend and frontend communicate using the VMQ inter-VM message passing communications mechanism. One or more VMQ links must be declared in the platform device tree for each backend-frontend pair. Each VMQ link is used to export one or several devices.

The example below declares one VMQ link between VM2 and VM3 using the virtual link framework, and exporting 7 video devices.

```
&vm2_vdevs {

    vvideo_be: vvideo-be {
        compatible = "vvideo2";
        info = "be";
    };

};

&vm3_vdevs {

    vvideo-fe {
        peer-phandle = <&vvideo_be>;
        info = "fe ",
               "vvideo2=(s5p-mfc-dec0,6) ",
               "vvideo2=(s5p-mfc-enc0,7) ",
               "vvideo2=(s5p-mfc-dec-secure0,8) ",
               "vvideo2=(s5p-mfc-enc-secure0,9) ",
               "vvideo2=(s5p-mfc-enc-otf0,10) ",
               "vvideo2=(s5p-mfc-enc-otf-secure0,11) ",
               "vvideo2=(exynos5-scaler:m2m,50) ";
    };

};
```

In the *info* property on the frontend side, each

```
vvideo2=(device-name[,[original-node-number][,[frontend-node-number][,[async-mode]]]])
```

parameter specifies:

- one exported video *device-name*
- optionally, the *original-node-number* of backend video device
- optionally, the desired *frontend-node-number*, if different from backend value. V4L2 may allocate a higher value in case of numbering clash
- optionally, the desired async-mode: -1 (default policy, Scaler only), 0 (disabled) or 1 (enabled).

Asynchronous mode defers sending of some well-known ioctl() requests to backend in order to reduce the virtualisation overhead, while at the same time reporting success to issuing process. The bitmap of requests which are treated asynchronously is dynamically adjustable through the */sys/kernel/debug/vvideo2-fe/async_mask* file and also visible through the */proc/nk/vvideo2-fe* file.

Request	Bit
VIDIOC_S_CTRL	0x1
VIDIOC_STREAMON	0x2
VIDIOC_REQBUFS count=0	0x4
VIDIOC_S_CROP	0x8

For hot-pluggable devices, there is no fixed *original-node-number* so the *device-name* can be specified alone or with a *frontend-node-number*, as e.g. in:

```
vvideo2=(Philips 740 webcam,,60)
```

The *fe* string can be changed to *fe*,msg-count if the default maximum number of parallel requests (4 times the number of exported devices) is not sufficient, due to devices being used at the same time by many processes. In the configuration above, up to 28 requests (4 requests times 7 devices) are allowed in parallel.

For VMQ protocol reasons, any *msg-count* value passed or computed internally is rounded up to next power of two if it is not already a power of two. This is visible in */proc/nk/vmq*.vvideo2-be and */proc/nk/vmq*.vvideo2-fe files.

Note

Requests other than *VIDIOC_DQBUF* are usually short-lived, so total capacity depends mostly on how many sleeping *VIDIOC_DQBUF* requests are issued in parallel.

1.2.27.6 SEE ALSO

- VMQ(4D) - a driver internally used for communications between virtual machines
- VION(4D) - a driver internally used to share DMABUFs between virtual machines
- VFENCE2(4D) - a driver internally used to share DMA fences between virtual machines
- VGKI(4D) - a driver internally used by vvideo2 backend for Generic Kernel Image compatibility

1.2.28 VTHERMAL(4D)

1.2.28.1 Cross References

Related Documents
Manual Page

1.2.28.2 NAME

vthermal — Virtual thermal sensor

1.2.28.3 SYNOPSIS

The virtual thermal (vthermal) backend and frontend drivers, when paired together, allow a user processes running on different VMs to request a temperature measurement for a particular physical thermal zone.

1.2.28.4 FEATURES

Under Linux, thermal zone devices are typically represented by nodes in the device tree. For a general overview, please refer to `Linux thermal framework tree binnding.`

Virtual thermal backend driver's responsability is to serve the requests comming from the frontend driver and then to initiate thermal measurements. It can be enabled in the Linux kernel configuration:

Device Drivers -> VLX virtual device support -> Virtual thermal sensor backend

Virtual thermal frontend driver's responsability is to register thermal sensors inside existing thermal zones (previously registered by the thermal core driver) and to serve requests for thermal data comming from user space processes. It can be enabled in the Linux kernel configuration:

Device Drivers -> VLX virtual device support -> Virtual thermal sensor frontend

This configuration depends on the following:

Device Drivers -> Generic Thermal sysfs driver

Virtual thermal backend and frontend drivers communicate using VRPC. One VRPC link must be declared in the platform device tree for each thermal sensor registered by the frontend driver. The example below declares one vrpc link (VM2, VM3) using the virtual link framework (see vrpcmanual page for more details).

```
&vm2_vdevs {
   vthermal_be: vthermal@be {          // vthermal backend
        compatible = "vrpc";           // uses generic vRPC protocol
        server;
        info       = "vthermal";       // driver name
   };
};

&vm3_vdevs {
    vthermal@fe {                          // VM3 vthermal frontend
        peer-phandle = <&vthermal_be>;     // peer vLINK
        client;                            // front-end end point
        info         = "vthermal";         // driver name
    };
};
```

In the next example a configuration with multiple links is given.

```
&vm2_vdevs {
   vthermal_be_cpu: vthermal@be_cpu {   // vthermal backend
        compatible = "vrpc";        // uses generic vRPC protocol
        server;
        info       = "vthermal";    // driver name
   };
   vthermal_be2_gpu: vthermal@be_gpu {  // vthermal backend
        compatible = "vrpc";        // uses generic vRPC protocol
        server;
        info       = "vthermal";    // driver name
   };
};

&vm3_vdevs {
    vthermal@fe {                             // VM3 vthermal frontend
        peer-phandle = <&vthermal_be_cpu>;    // peer vLINK
        client;                               // front-end end point
        info         = "vthermal";       // driver name
    };
    vthermal@fe {                             // VM3 vthermal frontend
        peer-phandle = <&vthermal_be_gpu>;    // peer vLINK
        client;                               // front-end end point
        info         = "vthermal";            // driver name
    };
};
```

Below is given a typical example of a single thermal zone device and associated thermal sensor.

```
    vltsensor: vltsensor {
        compatible = "vl,thermal-sensor";
        #thermal-sensor-cells = <0>;
    };

    thermal-zones {
        therm_zone0: therm_zone0 {
                polling-delay-passive = <0>; /* milliseconds */
                polling-delay = <0>; /* milliseconds */

                thermal-sensors = <&vltsensor>;
        };
    };
```

This parent "thermal-zones" node acts as a container for all thermal zone devices. Each child node inside it describes a single thermal zone. The "thermal-sensors" property refers to a specified sensor. All sensors must contain a compatible property set to "vl,thermal-sensor".

In the domain where the fronend drivers run the init procedure of the thermal core driver reads the configuration and registers thermal zone devices if any. Then the init procedure of vthermal frontend driver registers sensors inside these previously registered zones. The mapping between zones inside different VMs is done by the name of the thermal zone. During a preparation of a request the frontend driver fills the PMEM region with the name of the zone in interest. When notified the backend driver reads the name of this zone from the PMEM region and then initiates a lookup for this particular zone.

Below is given a configuration with multiple thermal zones inside the frontend driver's domain.

```
    vltsensor_cpu: vltsensor_cpu {
        compatible = "vl,thermal-sensor";
        #thermal-sensor-cells = <0>;
    };

    vltsensor_gpu: vltsensor_gpu {
        compatible = "vl,thermal-sensor";
        #thermal-sensor-cells = <0>;
    };
```

```
thermal-zones {
    therm_zone0: therm_zone0 {
            polling-delay-passive = <0>; /* milliseconds */
            polling-delay = <0>; /* milliseconds */

            thermal-sensors = <&vltsensor_cpu>;
    };

    therm_zone1: therm_zone1 {
            polling-delay-passive = <0>; /* milliseconds */
            polling-delay = <0>; /* milliseconds */

            thermal-sensors = <&vltsensor_gpu>;
    };
};
```

In the given example above there are two thermal zones with names as follows:

- first thermal zone - "therm_zone0"
- sencond thermal zone - "therm_zone1"

These names must match the ones of the thermal zones inside SYS domain.

Below is given the thermal tree of zone0 in sysfs.

```
root@android:/ # ls /sys/class/thermal/thermal_zone0/
integral_cutoff
k_d
k_i
k_po
k_pu
mode
offset
passive
policy
power
slope
subsystem -> ../../../../class/thermal
sustainable_power
temp
type
uevent
root@android:/ #
```

Type field represents the name of the thermal zone.

```
root@android:/ # cat /sys/class/thermal/thermal_zone*/type
therm_zone0
therm_zone1
therm_zone2
therm_zone3
root@android:/ #
```

1.2.28.5 SEE ALSO

vrpc

Linux thermal framework sysfs API.

Linux thermal framework device tree bindings.

1.2.29 VWATCHDOG(4D)

1.2.29.1 Cross References

Related Documents
Manual Page

1.2.29.2 NAME

vwatchdog — Virtual watchdog driver

1.2.29.3 SYNOPSIS

Virtual watchdog driver vwatchdog is implemented on top of the Hypervisor watchdog device and it exports control and monitoring interface to it which is compatible with the Linux Watchdog API.

1.2.29.4 FEATURES

First thing in case of usage of vwatchdog is enabling the driver inside the Linux kernel configuration. It also requires some additional options related to the Linux Watchdog API.

```
CONFIG_WATCHDOG_CORE=y
CONFIG_WATCHDOG=y
CONFIG_VLX_VWATCHDOG=y
```

Second thing that is required is the Hypervisor property configuration. Based on this configuration the vwatchdog driver instantiates a device and loads an initial timer configuration. Also this property attributes and structure define the ownership and access rights to the watchdog. Format of the configuration is following:

```
    nk.vm.<vmid>.wdt.<name>.config {
        size         = <4>; // 4 x 32-bits integers
        allow-set    = "<vmid>"; // Owner VM, depends on security policy
        allow-get    = "<vmid>"; // Owner VM, depends on security policy
        allow-notify = "<vmid>"; // Owner VM, depends on security policy
                       // The watchdog configuration: the timer counter values (in us)
                       // and the working mode
            // LOAD: 500ms
            // BOOT: 10s
            // BARK: disabled = 0,
            // MODE: NK_VWDT_MODE_VM_NOTIFY = 2
        cell-value        = <LOAD>,<BOOT_LOAD>,<BARK>,<MODE>
    };

    nk.vm.<vmid>.wdt.<name>.counter {
        size         = <NK_VWDT_COUNT_SIZE>;
        allow-set    = "<vmid>"; // Owner VM to pat the watchdog.
    };

    nk.vm.<vmid>.wdt.<name>.state {
        size         = <NK_VWDT_STATE_SIZE>;
        allow-set    ; // Typically nobody, read-only property
        allow-get    = "<vmid>"; // Owner VM
        allow-notify = "<vmid>"; // Owner VM
    };
```

vwatchdog configuration contains three properties:

- config
- counter

- state

Name of the property contains the VM id which will instantiate the device:

```
nk.vm.<vmid>.wdt.<name>.<prop-name>
```

Typical configuration:

```
   nk.vm.3.wdt.cl0.config {
       size          = <16>; // 4 * 32 bits = 16 bytes
       allow-set     = "3";
       allow-get     = "3";
       allow-notify ;
            // LOAD: 40s
            // BOOT: 41s
            // BARK: disabled = 0,
            // MODE: NK_VWDT_MODE_VM_REBOOT = 1
       cell-value = <(40*1000*1000)>,<(41*1000000)>,<0>,<1>;
   };
   nk.vm.3.wdt.cl0.counter {
       size          = <4>; // 4 bytes
       allow-set     = "3"; //
   };
   nk.vm.3.wdt.cl0.state {
     size          = <4>; // 4 bytes
     allow-set     = ""; // read-only
     allow-get     = "3"; //
     allow-notify = "3"; //
   };
// == probe configuration
   nk.probe.cpu.0.enable {
     size          = <4>; // 4 bytes
     allow-set     = "*"; // all
     allow-get     = "*"; // all
     allow-notify;    // nobody
   };
```

When the driver is loaded it will create a device which is accessible by device file /dev/watchdogX.

The driver supports following IOCTLs from the Linux Watchdog API:

1. WDIOC_KEEPALIVE
2. WDIOC_SETTIMEOUT
3. WDIOC_GETTIMEOUT
4. WDIOC_SETPRETIMEOUT
5. WDIOC_GETPRETIMEOUT

The 'Magic Close' feature is supported by the driver, but it will not take affect since the WDOG_NO_WAY_OUT option is set.

For more information about the Linux Watchdog API, please refer the following document: Linux Watchdog API

1.2.29.5 SEE ALSO

Linux Watchdog API

1.2.30 VSMQ(4D)

1.2.30.1 NAME

vsmq — Virtual Simple Message Queue communication layer for virtual drivers.

1.2.30.2 SYNOPSIS

#include <vsmq.h>

This virtual Simple Message Queue (VSMQ) layer provides a generic fixed size message communication library for virtual drivers.

1.2.30.3 FEATURES

This library is launched as a Linux module through the **insmod** command. The module can also be embedded into the kernel image.

1.2.30.4 DESCRIPTION

The virtual Simple Message Queue library provides a bi-directional point-to-point communication channel allowing to asynchronously exchange fixed size messages between two VMs using a virtual link. Each virtual link has a transmit and receive queue located into the shared persistent memory (PMEM). The library interface is composed of primitives managing messages and notification call back routines. The library is hardened against unexpected behaviour of the peer VM and therefore it can safely be used for the communication with a non-trusted VM.

1.2.30.5 EXTENDED DESCRIPTION

The driver creates a file in the /proc file system for each uni-directional virtual link.

The file corresponding to transmission queue is named as following:

/proc/nk/vsmq-**name**-tx-**vmid**.**linkid**

The file corresponding to reception queue is named as following:

/proc/nk/vsmq-**name**-rx-**vmid**.**linkid**

where:

- **name** - the name of virtual link (**compatible** string of the virtual link node)
- **vmid** - the peer (remote) VM identifier
- **linkid** - local virtual link identifier (see vsmq_link_id)

These files provide incremental statistics as shown in the following example.

```
$ cat /proc/nk/vsmq-vgpu-arb-comm-rx-3.8fc002c0
Link Peer      Rx-Msg    Rx-Free     Rx-IRQ Rx-Sysconf Tx-Sysconf Reset Name
   1    3   19510810   19510810   19510760        100          2     1 vgpu-arb-comm

$ cat /proc/nk/vsmq-vgpu-arb-comm-tx-3.8fc00300
Link Peer      Tx-Msg   Tx-Alloc     Tx-IRQ Rx-Sysconf Tx-Sysconf Reset Name
   1    3   19255257   19255257   19255258        102          2     1 vgpu-arb-comm
```

See detailed Virtual SMQ API description.

1.2.31 VMBOX(4D)

1.2.31.1 NAME

vmbox — Virtual Message Box communication layer for virtual drivers.

1.2.31.2 SYNOPSIS

#include <vmbox.h>

This virtual Message Box (VMBOX) layer provides a generic fixed size message communication library for virtual drivers.

1.2.31.3 FEATURES

This library is launched as a Linux module through the **insmod** command. The module can also be embedded into the kernel image.

1.2.31.4 DESCRIPTION

The virtual Message Box library provides a star topology communication infrastructure allowing to asynchronously exchange fixed size messages among multiple VMs. The communication layer is based on the virtual Simple Message Queue library which provides a bi-directional point-to-point communication channel. The virtual Message Box library is particularly useful in order to create a communication infrastructure between a single back-end driver (server) and multiple front-end drivers (clients). For instance, a single back-end driver can be a central arbiter which orchestrates a mediated pass-through virtualization solution by exchanging messages with multiple para-virtualized device drivers.

The library interface is composed of primitives managing messages and notification call back routines. The library is hardened against unexpected behaviour of the peer VM and therefore it can safely be used for the communication with a non-trusted VM.

1.2.31.5 EXTENDED DESCRIPTION

See detailed Virtual MBOX API description.

1.2.32 SVEC(4D)

1.2.32.1 Cross References

Related Documents
Manual Page

1.2.32.2 NAME

svec driver — Simple Virtual Event Controller driver

1.2.32.3 SYNOPSIS

The svec-driver implements a general purpose hypervisor to kernel event notification mechanism, similarly to a interrupt controller driver.
The svec-driver is of a special kind since it only implements a kernel API (it cannot be used from user space).
It is, for now, only used by the vproperty subsystem to notify the vproperty driver that a property content has been changed, but it is intended to be deployed widerly as needed.

1.2.32.4 FEATURES

The svec-driver is tightly coupled to its hypervisor svec-device counterpart, with which it implements the following features:

- svec-device instances are exposed to VMs by the hypervisor in accordance with the configuration specified in the hypervisor device tree. A svec-device instances is uniquely identified in a VM by its name, a VM can host several svec-device instances with different names. The svec-driver controls all the svec-device instances of the VM.
- svec-driver exposes events (up to 1024), to which user drivers can attach handler functions and a cookie parameter.
- when the hypervisor posts an event to a svec-device instance, the handler functions attached to this event are called with their associated cookie provided as parameter.
- user drivers can attach any number of handler functions to events, the same handler function can be attached several times (to the same or to different events).
- user drivers can mask and unmask events (actually attachments) to protect against handlers execution (this is similar to interrupt masking).
- svec-driver exposes an API (svec_evt_attach(), svec_evt_detach(), svec_evt_mask(), svec_evt_unmask()) whose primitives can be called indifferently from within or from outside handler functions.

1.2.32.4.1 CONFIGURATION

svec-device instances creation is controlled by "vl,hypevent.v1" compatible 'vdevs' nodes in the hypervisor device tree.
The server endpoint node info property cointains the name of the svec-device instance to be exposed. Example: the configuration below illustrates how 'vproperty' svec-device instances are created in VM#2 and VM#3.

```
&vm1_vdevs {
    ...
    hypevent_vproperty: hypevent@vproperty {
        compatible = "vl,hypevent.v1";
        info = "vproperty";
        #clone = "auto";
        server;
    };
    ...
};

&vm2_vdevs {
    ...
    hypevent@vproperty {
```

```
        peer-phandle = <&hypevent_vproperty>;
    client;
    };
    ...
};

&vm3_vdevs {
    ...
    hypevent@vproperty {
        peer-phandle = <&hypevent_vproperty>;
    client;
    };
    ...
};
```

Note

"vl,hypevent.v1" compatible nodes always have their server endpoint in the hypervisor VM (i.e. the VM#1), their client endpoint in a non-hypervisor VM (VM::i where i is in [2, 31]).

1.2.32.5 PARAMETERS

N/A.

1.2.32.6 SEE ALSO

N/A.

1.2.33 VLX-CPU-HANDOVER(4D)

1.2.33.1 Cross References

Related Documents
Manual Page

1.2.33.2 NAME

vlx-cpu-hotplug — CPU Handover virtual driver

1.2.33.3 SYNOPSIS

The CPU Handover virtual driver vlx-cpu-hotplug processes the CPU Handover requests issued by the hypervisor through the set of exported vproperties which implement the CPU Handover API.

CPU Handover requests are translated to the guest OS CPU shutdown or CPU start commands through a call to the kernel appropriate methods for CPU hotplug management.

1.2.33.4 FEATURES

The CPU handover mechanism allows to create configurations where multiple VMs are assigned to a same physical CPU while ensuring that they cannot use it at the same time.

The ownership of a physical CPU can be changed dynamically while the system is running by interfacing with the CPU handover module through hypervisor properties.

If the CPU handover feature must be supported on a VM, it is necessary to enable the driver as a linux built-in driver within the configuration of the VM kernel.

```
CONFIG_VLX_CPU_HOTPLUG=y
```

1.2.33.5 CONFIGURATION

CPU dynamic dispatch is entirely controlled by the hypervisor. Each instance of the vlx-cpu-handover driver will cooperate with the hypervisor by bringing into play the guest OS kernel interface for CPU hotplug management in order to reflect the allocation or the withdrawal of one physical CPU by the hypervisor as part of a CPU handover transaction.

CPU Handover feature configuration is carried by the hypervisor device tree and consists in defining the set of physical CPUs which can be dynamically dispatched among the VMs.

```
&vm<vmid>_vcpus {
    #count = "*";
    affinity = "<vm-possible-cpus-list>";
    handover-affinity = "<vm-dynamically-allocatable-cpus-list>";
    migration = "<migration-mode>";
};
```

Configuration for one VM must includes the following parameters:

- vmid: identifer of the VM in the device tree.
- count: the number of all possible physical CPUs for the VM.
- affinity: the list of physical CPUs over which the VM can schedule its vCPUs.
- handover-affinity: the list of physical CPUs which can be granted to or withdrawn from the VM usage.

Notes:

- count must reference the number of all possible CPUs for the lifetime of the VM, whether available at VM start or later.
- <vm-dynamically-allocatable-cpus-list> must be a subset of <vm-possible-cpus-list>.
- <migration-mode> has an impact on vdriver used API to cooperate with the hypervisor, but not directly on the handover functionality which will behave in the same manner regarding the guest OS CPU management.
- handover-affinity can be present but with the value of '0' to indicate that the VM will in fact not actually support CPU handover.

Configured migration mode	vlx-cpu-handover mode
"off"	PINNED
"on"	FLOATING

Typical configuration:

```
&vm4_vcpus {
    #count = "*";
    affinity = ""1,2,3,5,6,7""; /* List of allowed physical CPUs */
    handover-affinity = "6,7";  /* Subset of CPUs subject to handover */
    migration = "off";          /* Pinned mode */
};
```

When the driver will start on the VM4, it will read the hypervisor exported property for handover, and will retieve and interpret the configuration above in the following way:

- The VM vCPUs will be mapped to allowed physical CPU on a one-to-one basis (pinned mode).
- The VM is allowed to schedule its vCPUs on the CPU1, CPU2, CPU3, CPU4, CPU5, CPU6, CPU7.
- The CPU6 and CPU6 cpus can be granted to or withdrawn from the VM4 usage according to handover requests coming from the hypervisor.

1.2.33.6 GUEST OS CONFIGURATION

As CPU handover rely on the ability of the guest OS to manage CPU hotplug, some constraints of the guest OS, for instance linux, must be respected when bringing he CPU Handover feature into play.

CPUs unavailable at boot time

If the guest OS must be aware of all the possible CPUs at boot time to dimension its internal resource management structures, then it may be necessary to signal that some CPUs, while possibly available in the futur of the VM life cycle, should be left offline at boot time.

Linux kernel typical configuration to restrain the set off booted CPUs:

```
boot_cpus=0,1,2,3
```

boot_cpus is the linux kernel init parameter that will define the set of CPUs that the kernel will have to take online at boot time.

```
- boot_cpus list is expressed from the guest OS point of view.
- correspondance between the physical CPUs and the guest OS CPUs is
  defined by the numbering of the hypervisor provided vCPUs.
```

correspondance between the physical CPUs and the guest OS CPUs

This is a hypervisor provided feature for the virtual platform binding configuration.

See: "Device virtualization reference manual - v11.2 - Public Edition", "Virtual CPU resources - "/vlm/vm<id>/vcpus/" nodes"

```
When the virtual CPUs migration is disabled, by default, the Hypervisor
uses an identical binding between the virtual and physical CPUs. In
particular, the Hypervisor binds the virtual CPU0 to the lowest physical
CPU ID from the affinity set. Then, the Hypervisor binds the virtual CPU1
to the lowest non bound physical CPU ID from the affinity set and so on.

Such a default identical mapping can be replaced with an custom one
defined by child nodes which name is starting with a fixed vcpu@ string
and followed by a virtual CPU ID. Such a node defines a binding for the
virtual CPU corresponding to the ID in the node name. The corresponding
physical CPU ID is defined by a cpu property in this node.

In the following example, the virtual CPUs 0 and 1 are cross mapped to the
physical CPUs 1 and 0 accordingly.

   &vm2_vcpus {

        migration = "off";

        vcpu@0 {
            cpu = <1>;   // vCPU0 -> CPU1
        };

        vcpu@1 {
            cpu = <0>;   // vCPU1 -> CPU0
        };

    };
```

1.2.33.7 USER SPACE DESCRIPTION

The vlx-cpu-handover virtual driver exports only monitoring API to the user space through the use of hypervisor properties.

1.2.33.8 /sys/nk/prop Entries

The following properties are setup by the hypervisor according to the CPU handover configuration defined in its device tree.

Userspace program can read these properties to monitor the state of CPU handover for the addressed VM <vmid>.

```
nk.cpus.online      /* Mask of system wide online physical CPUs. */

nk.vm.<vmid>.handover-affinity  /* Mask of the physical CPUs that the VM can share through handover. */

nk.vm.<vmid>.cpus   /* Mask of physical CPUs allocated to the VM. */
nk.vm.<vmid>.vcpus  /* Mask of virtual vCPUs allocated to the VM. */

nk.vm.<vmid>.vcpus-target   /* Mask of the vCPUs re-assigned to the VM. */

nk.vm.<vmid>.vcpus.online   /* Mask of the vCPUs currently online in the VM space. */
```

The virtual driver will test the actual instance of the CPU handover API for the VM it is running on.

Notes:

- If the nk.vm.<vmid>.handover-affinity property carries a value of zero, then the VM <vmid> will be considered as not supporting the CPU handover feature by userspace applications.
- If the nk.vm.<vmid>.vcpus-target propert carries a value of zero, then the VM <vmid> will be considered as managing floating vCPUs by userspace applications.

If no CPU Handover API instance, or inconsistent instance, is found at boot time by the vlx-cpu-handover driver then the driver will log an error status message through kernel log functions and will fallback to an idle mode where no handover request will be processed.

1.2.33.9 Processing hypervisor handover requests

1.2.33.9.1 SYNOPSYS

In **pinned** mode, the vlx-cpu-handover virtual driver will shutdown or start guest OS CPUs according to the value of **nk.vm.<vmid>.vcpus-target** property.

In **floating** mode, the vlx-cpu-handover virtual driver will shutdown or start guest OS CPUs according to the value of **nk.vm.<vmid>.cpus** property.

1.2.33.10 DESCRIPTION

The CPU hotplug virtual driver implements, within the guest OS instance running in a hypervised virtual machine, the necessary operation set which ensures the support of the CPU handover feature.

The CPU handover module requires cooperation from the GuestOS as the hypervisor cannot forcefully remove a VCPU from a GuestOS.

At a higher level, a governor entity decides of a new CPU dispatch among the various VMs of the system. This governor will require the hypervisor to perform the handover of addressed CPUs from some VMs to some others in order to match the CPU new dispatch scheme.

The hypervisor will in turn decompose these handovers of CPUs into some sequences of CPU withdrawal and CPU acquisition requests through the CPU handover API described in the `"CPU handover HLD"`.

The guest OS manages a set of CPUs, each of this guest CPU being mapped over one virtual CPU (vCPU) managed by the hypervisor, each vCPU allowing in turn the guest OS to actually access a physical CPU in order to get VM sandboxed software actual execution.

CPU handover vlx-cpu-handover virtual driver API is described in the low-level design `"CPU handover LLD"`.

1.2.33.11 Tuning and debugging API

CPU Handover virtual driver provides a tuning and debugging API based on linux filesystem debugfs framework.

See: `<debugfs>/vlx-cpu-hotplug_debugfs/no-action-on-cpus(5)`.

If CPU handover debugfs API must be used then it is necessary to activate it as an option of the built-in driver for the VM kernel.

```
CONFIG_VLX_CPU_HOTPLUG_DEBUG=y
```

1.2.34 VLX-PANIC-TRIGGER (4D)

1.2.34.1 NAME

vlx-panic-trigger — PANIC Trigger virtual driver

1.2.34.2 SYNOPSIS

The "panic trigger" driver aims to cause a `kernel panic` of the current Guest Operating System after a remote Guest Operating System has entered the "Kernel Panic" state.

This driver is used during system development phases, the idea is to force all Guest OSes and the Hypervisor to dump debugging information in persistent memory for later analysis when a Kernel Panic of one Guest OS occurs.

The debugging information is stored persistent memory area. It can be analysed online by using a debugging tools (like Trace32), it can be saved in flash during the next reboot for an later analysis.

1.2.34.3 IMPLEMENTATION DETAILS

The "panic trigger" mechanism is based on the Hypervisor property (also known as vproperty) feature.

This feature provides a publish/subscribe mechanism. An vproperty is a storage area within the Hypervisor space which can be referenced by a key. The hypervisor provides APIs allowing to set/get the property value. In addition a notification mechanism can be configured so that a Guest Operating system can be notified of the modification of the value. "vproperties" are configured by device tree nodes located in the Hypervisor device tree.

For instance,the "nk-panic-trigger" vproperty is configured as follow :

```
&hyp_property {
    nk.panic-trigger {
        size = <1>;
        allow-set = "*";
        allow-get = "*";
        allow-notify = "*";
        value = "0";
    };
};
```

In the above example, the "nk.panic-trigger" property is configured as follow:

- any VM can read or write the property,
- the key of the property is "nk.panic-trigger",
- the size of the value is "1" byte
- the default value is "0"
- any VM is notified when the value is changed.

At initialization time, the driver connects to the notification mechanism.

When a Guest OS panics for any reasons it sets the "nk.panic-trigger" vproperty with its Virtual Machine identifier.

Here is a snippet of Linux driver code performing such an operation:

```
static void hyp_early_panic(void)
{
        /*
         * Trigger system-wide panic by raising the panic trigger property.
         */
        uint8_t value = 0;
        size_t size = nkops.nk_prop_get("nk.panic-trigger",
                                        &value, sizeof(value));
        if ((size == 0) || ((size == sizeof(value)) && (value == 0))) {
                value = nkops.nk_id_get(); // get vm-id
                size = nkops.nk_prop_set("nk.panic-trigger",
                                         &value, sizeof(value));
                if (size != sizeof(value))
                        pr_notice("Could not initiate system-wide panic.\n");
                else
                        system_wide_panic = 1;
        } else {
                pr_notice("system-wide panic already initiated by VM%d\n",
                          value);
        }
}
```

Once the property is set, the Hypervisor notifies all other VM by triggering a "SYSCONF" interrupt in remote virtual machines.

In remote virtual machine, the "SYSCONF" interrupt handler (of the "vlx-panic-driver" driver) which has been registered at initialization times gets invoked and finally calls the **"panic"** routine of the Linux kernel.

1.2.34.4 CONFIGURATION

By default, the driver is configured. So it is enough to configure the Hypervisor property to activate this mechanism.

1.2.34.5 REMARKS

This mechanism is a security breach because it allows a non trusted Guest OS to make fail a trusted Guest OS. So the property "nk.panic-trigger" shall not be configured in production systems.

1.3 (5) Files and directories

/sys/nk(5)
/proc/nk(5)
<debugfs>/vlx-cpu-hotplug_debugfs/no-action-on-cpus(5)

1.3.1 /sys/nk(5)

1.3.1.1 NAME

/sys/nk — sysfs extension for exporting Hypervisor objects

1.3.1.2 DESCRIPTION

Paravirtualized Linux Guest OSes extend the traditional Linux sysfs file system with a directory */sys/nk* to provide access to some Hypervisor objecst.

1.3.1.3 Files and directories

1.3.1.3.1 /sys/nk/prop

This directory list the Hypervisor properties defined as part of the Hypervisor device tree. Each property is represented by a file whose name is the name of the property.

Most of the properties are read-only, however some of them are writable.

1.3.1.3.1.1 /sys/nk/prop/nk.build-id

This file provides the build identifier of the system. The string returned is NULL terminated but does not include a linefeed.

```
v11.1-142
```

1.3.1.3.1.2 /sys/nk/prop/nk.cpus.online

This binary file contains a 32 bits mask where bits set to 1 represent online physical CPUS. For example, a mask of 0x000000FF means that physical CPUS 0 to 7 are online.

1.3.1.3.1.3 /sys/nk/prop/nk.cpustats.hyp.enable

Deprecated /sys/nk/prop/nk.cpustats.hyp.enable is deprecated. See /sys/nk/prop/nk.monitoring.enable instead.

1.3.1.3.1.4 /sys/nk/prop/nk.error.ignore-alloc-after-boot

This is a read-only 32 bits value. If the value is 0, either binary (0x00) or the character '0' (0x30), any allocation error after boot will lead to a configuration panic of the system. If the value is different from 0x0 or 0x30, allocation errors occuring after boot will be ignored.

1.3.1.3.1.5 /sys/nk/prop/nk.error.ignore-non-allowed-io-sharing

This is a read-only 32 bits value. If the value is 0, either binary (0x00) or the character '0' (0x30), any unexpected IO memory sharing will lead to a configuration panic of the system. If the value is different from 0x0 or 0x30, unexpected IO memory sharing will be ignored.

1.3.1.3.1.6 /sys/nk/prop/nk.features.all

This a read-only 64 bits value. It is a mask of all features defined by the Hypervisor, whether they are currently enabled or disabled.

1.3.1.3.1.7 /sys/nk/prop/nk.features.all-str

This is a read-only string file. It provides the list of all features supported by the Hypervisor, whether they are are currently enabled or disabled. It is the human readable form of the previous property.

```
UART RAM_CONSOLE BALLOON CPU_USAGE DEBUG PCOV GCOV VDEV_VGIC_V2 VDEV_VGIC_V3 INSTRUMENT HTFVBR TIMING TIMING_T
```

1.3.1.3.1.8 /sys/nk/prop/nk.features.off

This is a read-only 64 bits value. It is a mask of all features currently disabled in the Hypervisor. Bits set to 1 correspond to disabled features.

1.3.1.3.1.9 /sys/nk/prop/nk.features.off-str

This is a read-only string file. It provides the list of all features currently disabled in the Hypervisor. It is the human readable form of the previous property.

```
BALLOON DEBUG PCOV GCOV VDEV_VGIC_V3 INSTRUMENT HTFVBR TIMING TIMING_TRACE PM_CPU_LPM PM_WAKEUP_IRQ VM_PM VDEV
```

1.3.1.3.1.10 /sys/nk/prop/nk.features.on

This is a read-only 64 bits value. It is a mask of all features currently enabled in the Hypervisor. Bits set to 1 correspond to enabled features.

1.3.1.3.1.11 /sys/nk/prop/nk.memory.heap.free

This is a read-only binary file. It provides the number of unused bytes in Hypervisor memory heap.

See also

/proc/nk/memstat

1.3.1.3.1.12 /sys/nk/prop/nk.memory.heap.used

This is a read-only binary file. It provides the number of used bytes in Hypervisor heap memory.

See also

/proc/nk/memstat

1.3.1.3.1.13 /sys/nk/prop/nk.memory.pmem.free

This is a read-only binary file. It provides the number of unused bytes in Hypervisor persistent memory.

See also

/proc/nk/memstat

1.3.1.3.1.14 /sys/nk/prop/nk.memory.pmem.used

This is a read-only binary file. It provides the number of used bytes in Hypervisor persistent memory.

See also

/proc/nk/memstat

1.3.1.3.1.15 /sys/nk/prop/nk.features.on-str

This is a read-only string file. It provides the list of all features currently enabled in the Hypervisor. It is the human readable form of the previous property.

```
UART RAM_CONSOLE CPU_USAGE VDEV_VGIC_V2 PM_CPU_HOTPLUG PM_S2R VDEV_FIXUP VDEV_PSCI VDEV_ARMVX_TIMER VDEV_NULL
```

A feature can only appear in the "on" list or in the "off" list. It cannot be present in both list simultaneously.

1.3.1.3.1.16 /sys/nk/prop/nk.monitoring.enable

This is a read-write 32 bits value. It enables to activate or deactivate monitoring features such as:

- Hypervisor events statistics
- CPU usage statistics
- Hypervisor events log
- VCPU registers snapshot

This are the ways to modify the monitoring:

- Writing 0 (0x00 or 0x30) activates 'Hypervisor events statistics'.
- Writing 1 (0x01 or 0x31) activates both 'Hypervisor events statistics' and 'CPU usage statistics'.
- Writing 2 (0x02 or 0x32) activates 'Hypervisor events statistics', 'Hypervisor events log' and 'VCPU registers snapshot'.
- Writing 3 (0x03 or 0x33) activates both 'Hypervisor events statistics' and 'VCPU registers snapshot'.
- Writing 'P' disactivates monitoring.

'Hypervisor events statistics', 'CPU usage statistics' and 'Hypervisor events log' can be retrieved through the /proc/nk(5) files. 'VCPU registers snapshot' can be retirved through the sysnk_nk_vm_id_cpus files.

1.3.1.3.1.17 /sys/nk/prop/nk.version.bsp

This is a read-only string file. It returns the string identifying the Board Support Package of the system.

1.3.1.3.1.18 /sys/nk/prop/nk.version.bsp-build-date

This is a read-only string file. It returns the date and time when the BSP has been generated.

```
Tue Feb 26 08:54:13 UTC 2019
```

1.3.1.3.1.19 /sys/nk/prop/nk.version.bsp-compiler

This a read-only string file. It returns the identification of the compiler used to build the BSP.

```
real-aarch64-linux-android-gcc (GCC) 4.9.x 20150123 (prerelease)
```

1.3.1.3.1.20 /sys/nk/prop/nk.version.build-date

This is a read-only string file. It returns the date and time when the Hypervisor has been built.

```
Tue Feb 26 08:51:38 UTC 2019
```

1.3.1.3.1.21 /sys/nk/prop/nk.version.compiler

This a read-only string file. IT caontains the identifier of the compiler used to build the Hypervisor.

```
real-aarch64-linux-android-gcc (GCC) 4.9.x 20150123 (prerelease)
```

1.3.1.3.1.22 /sys/nk/prop/nk.version.hypervisor

This a read-only string file. It contains the identifier of the version of the Hypervisor.

```
heads/products/v11.1/master-0-ge4bca79
```

1.3.1.3.1.23 /sys/nk/prop/nk.vlm-dtb

This read-only file provides the "Device Tree Blob" that was built at configuration time. This is described within the "Hypervisor Reference Manual > Configuration > Hypervisor DEvice Tree bindings".

1.3.1.3.1.24 /sys/nk/prop/nk.vm.[id].cpus

This is a read-only 32 bits value. It is a bit mask representing the list of physical CPUS on which the VCPUS of the VM [id] can run. If bit *i* is set to 1, the VM may use the corresponding physical CPU. Otherwise the physical CPU is not usable by the VM.

The CPU affinty defined in the "/vlm/vm[id]/vcpus/" node of the device tree impacts the value of this file.

1.3.1.3.1.25 /sys/nk/prop/nk.vm.[id].state.paused

This is a read-only 32 bits value. The bit 0 indicates whether the VM is "paused" (bit is set to 1) or "playing" (bit is set to 0). The remaining bits provide a value to count the number of times the VM has transitioned from "paused" to "playing" state.

1.3.1.3.1.26 /sys/nk/prop/nk.vm.[id].state.running

This is a read-only 32 bits value. The bit 0 indicates whether the VM is "running" (bit is set to 1) or "stopped" (bit is set to 0). The remaining bits provide a value to count the number of times the VM has transitioned from "stopped" to "running" state.

1.3.1.3.1.27 /sys/nk/vlinks

This directory holds a sub-tree, where regular files are of the form:

- endpoints/client/[vdriver]/[nb]/vm[id1]/vm[id2]/info
- endpoints/server/[vdriver]/[nb]/vm[id1]/vm[id2]/info

Many vdrivers in Linux Guest OSes rely on vlinks. A vlink end-point is used as a client or as a server. If several instances of the virtual device managed by a given vdriver are created, the value [nb] will identify the instance of the device. vm[id1] and vm[id2] identify the two VMs connected.

The info file is a read-only string. Its value is the value of the info property in the corresponding device tree node. This value is used by the vdriver at initialization time.

1.3.1.4 NOTES

1.3.1.5 SEE ALSO

/proc/nk(5)

1.3.2 /proc/nk(5)

1.3.2.1 NAME

/proc/nk — virtualization information pseudo-filesystem

1.3.2.2 DESCRIPTION

The **proc/nk** directory is a subdirectory of the */proc* filesystem. It provides access to Harman virtualization information. The entries in this directory are automatically created by the Guest OS. However the presence of these entires may depend upon the configuration options of the Linux kernel relative to the virtualization modules (for example such as the VLX_VINFO module).

1.3.2.2.1 Overview

Underneath */proc/nk*, there are the following general groups of files and subdirectories. Some other files or directories related to virtual devices may also appear, they are described in their respective (4D) man pages.

- /proc/nk/cpu/[id]
- /proc/nk/focus
- /proc/nk/history
- /proc/nk/id
- /proc/nk/last
- /proc/nk/log-[id]
- /proc/nk/memstat
- /proc/nk/monitoring
- /proc/nk/pcpustats
- /proc/nk/props
- /proc/nk/regions
- /proc/nk/restart
- /proc/nk/state
- /proc/nk/vmresume
- /proc/nk/vmstart
- /proc/nk/vmstop
- /proc/nk/vmsuspend
- /proc/nk/xirqmap
- /proc/nk/counter-timer-offset

1.3.2.2.2 Files and Directories

1.3.2.2.2.1 /proc/nk/cpu/[id]

There is one such directory per CPU of the underlying platform. CPU numbering starts at 0. A platform with 2 clusters with 4 cores each shows CPU from 0 to 7. Currently, the cpu[id] directory contains a single *pmon* subdirectory. This *pmon* directory has 2 subdirectories: *ascii* and *raw*. They provide access to the same contents. Files in the *raw* directory are binary files. The data structures are defined in the *vlx/include/perfmon.h* header file. Files in the *ascii* directory are ascii files.

1.3.2.2.2.2 /proc/nk/cpu/[id]/pmon/ascii/cpustats

Provides some statistical information regarding the cpu usage.

```
CPU 0 Consumption Statistics
PERIOD: 52052448
IDLE :  51990115

OS 00:  51990115
OS 01:  8192
OS 02:  54141
OS 03:  0
MISC0:  1
MISC1:  1
MISC2:  1
```

All times are expressed in Hypervisor ticks. Ticks must be divided by the frequency to obtain time expressed in seconds.

The *PERIOD* field gives the duration of the period used for collecting these statistics. The period is computed since the last time the statistics have been retrieved.

The *IDLE* field gives the time spent idling during the period.

The *OS 00* field gives also the idle time.

The *OS 01* field gives also the time during which the Hypervisor has been busy.

Then the busy time of that CPU is split among the various Guest OSes. The sum of all *OS ii* fields is normally equal to the period time.

1.3.2.2.2.3 /proc/nk/cpu/[id]/pmon/ascii/sysinfo

This file provide some general information such as the version of the performance analysis module, the frequency of the Hypervisor timer and the number of Guest OSes configured.

```
VLX Performance Analysis v1.0
Timer frequency : 26000000 Hertz
Number of OSes  : 3
```

1.3.2.2.2.4 /proc/nk/focus

The hypervisor property "nk.focus" contains the VMid of the VM which owns the input (touchscreen) and display devices.

This property is used when a single display and touch screen is shared by multiple VMs. In that case, the input and the display back-end drivers uses this property to know which VM owns those devices.

Reading the /proc/nk/focus file returns the ASCII encoded vmid of the "nk.focus" property.

Writing [VMid] to the /proc/nk/focus file sets the "nk.focus" hypervisor property with [VMid] and cause the input event back-end driver to send input events to the assigned [VMid] and cause the display back-end driver to display the virtual screen of the VM [VMid].

1.3.2.2.2.5 /proc/nk/history

This read-only file contains the console output of the hypervisor. The console of all Guest OSes are redirected to that common console. The output is stored in a history buffer. Output of the various Guest OSes is differentiated based on their color codes defined in their */vlm/vm[id]/console/* configuration parameter.

Reading the history file does not remove read lines from the content.

Reading '/proc/nk/history' file beyond the last available character returns EOF. The exact same console output content is also available in the '/dev/vlx-console' file. The only difference is that reading '/dev/vlx-console' file beyond the last available character blocks until at least a character is available.

The '/proc/last_kmsg' file may contain the Hypervisor console output collected before the last warm reboot.

1.3.2.2.2.6 /proc/nk/id

This read-only file contains a string with the number of the current Guest OS.

Beware that the returned string is not null terminated!

1.3.2.2.2.7 /proc/nk/last

This read-only file contains a string with the number of the last Guest OS created on the Hypervisor. Hence it is the greatest identifier of the configured Guest OSes.

Beware that the returned string is not null terminated!

1.3.2.2.2.8 /proc/nk/log-[id]

The *log-[id]* files provide access to the circular log buffers managed by the Hypervisor. The number of log buffers is defined as part of the Hypervisor configuration. These buffers are filled by the Hypervisor and the Guest OSes for debug purpose.

Reading a *log-[id]* file past the end of the file is non-blocking and returns EOF.

The '/dev/vlx-log-[id]' files provide the same content as the '/proc/nk/log-[id]' files. Reading after the last character of these files blocks the reader until at least one character is available.

1.3.2.2.2.9 /proc/nk/memstat

This read-only file provides current memory usage by the Hypervisor.

```
# cat /proc/nk/memstat
               SIZE        FREE        USED
HEAP       32747520     8855424    23892096
PMEM       33554432    27848704     5705728
```

Information about the following types of memory is listed:

- **HEAP** these are Hypervisor heap memory usage statistics in bytes.
- **PMEM** these are Hypervisor persistent memory usage statistics in bytes.

Memory status can also be accessed through following files:

- /sys/nk/prop/nk.memory.heap.free
- /sys/nk/prop/nk.memory.heap.used
- /sys/nk/prop/nk.memory.pmem.free
- /sys/nk/prop/nk.memory.pmem.used

1.3.2.2.3 /proc/nk/monitoring/event-log/filter-[set|clr]

1.3.2.2.4 Reading event-log filter state

The provided interface splits the filter state between the two managed procfs entries.

Reading the "nk/monitoring/event-log/filter-set" procfs entry returns a string listing by their names the events which are set in the event-log filter for monitoring selection.

Reading the "nk/monitoring/event-log/filter-clr" procfs entry returns a string listing by their names the events which are cleared from the event-log filter.

Set events will be monitored and recorded to the event-log ring buffers.
Cleared events will be filtered out and not stored in the event-log ring buffers.
Each event is designated by its name in lower-case characters.

Only the user event names are composed from a radix and their number.

Event identifer	Event name in "nk/monitoring/event-lo
NK_EVT_ID_CPU_START	cpu_start
NK_EVT_ID_CPU_STOP	cpu_stop
...	...
NK_EVT_ID_IRQ_POST	cpu_post
...	...
NK_EVT_ID_IRQ_THROTTLE	cpu_throttle
...	...
NK_EVT_ID_RANGE_USER_BASE	usr_0
NK_EVT_ID_RANGE_USER_BASE+1	usr_1
...	...
NK_EVT_ID_RANGE_USER_BASE+7	usr_7

For instance, the 160-bit long bit-mask value of:

[0x00000000 0x00000000 0x00000000 0x00000010 0x000000000080]

Will correspond to the following string read from "nk/monitoring/event-log/filter-set", with only "irq_post" and "irq_throttle" events allowed and all other monitoring events filtered out.

irq_post irq_throttle

The same bit-mask vlaue will correspond to the following string read from "nk/monitoring/event-log/filter-clr". All events but "irq_post" and "irq_throttle" being listed.

cpu_start cpu_stop intr_enter intr_leave intr_el2_enter
intr_el2_leave irq_select cpu_to_el3 cpu_from_el3
cpu_idle_enter cpu_idle_leave trap_enter trap_leave
hyp_call vcpu_to_el3 vcpu_from_el3 vcpu_start vcpu_stop
vcpu_suspend_request vcpu_resume_request vdev_load vdev_store
virq_assert vcpu_switch_out vcpu_switch_in vcpu_idle_enter
vcpu_idle_leave vm_stop vm_start vm_suspend_request
vm_resume_request vm_loading vm_loaded vm_fatal irq_spurious
vm_s2r_enter vm_s2r_leave s2r_enter s2r_leave
usr_0 usr_1 usr_2 usr_3 usr_4 usr_5 usr_6 usr_7

1.3.2.2.5 Setting event-log filter state by writing "nk/monitoring/event-log/filter-∗" procfs entries

The vlx-event-log driver allows an application to change the state of the hypervisor event log filtering by:

- writing the name of each event to be selected to the procfs entry "nk/monitoring/event-log/filter-set".
- writing the name of each event to be filtered out to the procfs entry "nk/monitoring/event-log/filter-clr".

The list of each selected event is reported through the "filter-set" entry and the list of each filtered out event is reported through the "filter-clr" entry, as these lists are obtain from the reading entries in the same format than expected for the writing entries, then:

- all events can be excluded by copying the list of currently selected events to the list of excluded events:

```
cat nk/monitoring/event-log/filter-set > nk/monitoring/event-log/filter-clr
```

- all events can be selected by copying the list of currenlty excluded events to the list of selected events:

```
cat nk/monitoring/event-log/filter-clr > nk/monitoring/event-log/filter-set
```

- as an event is either set or excluded, its name is always present in one of the "filter-set" or "fitler-clr" lists. The list of all the known events can then be obtained by concatenating these two lists, and each event name will be reported only once:

```
cat nk/monitoring/event-log/filter-set nk/monitoring/event-log/filter-clr
```

If the monitoring application wants to restricts the set of recorded events to "irq_post" and "irq_throttle", it can:

- write the name of each currently selected events to the 'filter-clr' entry, all events will then be filtered out.

```
cat nk/monitoring/event-log/filter-set > nk/monitoring/event-log/filter-clr
```

- write the following string to the "nk/monitoring/event-log/filter-set" procfs entry:

irq_post irq_throttle

All other events will be filtered out.

If the monitoring application wants to add the recording of the "hyp_call" and "usr_2" events to the monitoring filter, it then can write the following string to the "nk/monitoring/event-log/filter-set" procfs entry:

hyp_call usr_2

if the monitoring applicatoin wans to filter out the recording of the "usr_2" event, it can then write the following string to the "nk/monitoring/event-log/filter-clr" procfs entry:

usr_2

Note that an application can always access to the hypervisor exported property "nk.event-log.filter" to manipulate the bit-mask representation of the monitoring filer if it is more convenient, for instance in order to save and to restore the filter state.

1.3.2.2.5.1 /proc/nk/monitoring/vl,evt-log-0/cpu-[id]

There is one event log file per physical CPU. Each file gives access to a circular buffer of events which occurred on the CPU. Internally, events are represented as fixed size records. Records are 32 bytes long and are structured as follows: timestamp (8 bytes), payload (20 bytes), cpu id (1 byte), vm id (1 byte), vcpu id (1 byte), event type (1 byte).

The payload depends on the event type.

There are 35 event types:

- CPU_START: The current physical CPU is going online.
- CPU_STOP: The current physical CPU is going offline.
- INTR_ENTER: Processing of an interrupt taken from EL0/EL1 started.
- INTR_LEAVE: Processing of an interrupt taken from EL0/EL1 finished.
- INTR_EL2_ENTER: Processing of an interrupt taken from EL2 started.
- INTR_EL2_LEAVE: Processing of an interrupt taken from EL2 finished.
- IRQ_SPURIOUS: A spurious interrupt request has been detected.
- IRQ_SELECT: The interrupt being processed has been identified.
- IRQ_POST: An interrupt is being posted to a virtual CPU.
- CPU_TO_EL3: Execution of a SMC instruction started.
- CPU_FROM_EL3: Execution of a SMC instruction finished.
- CPU_IDLE_ENTER: The current physical CPU enters idle state from the hypervisor.
- CPU_IDLE_LEAVE: The current physical CPU resumes from idle state in the hypervisor.
- TRAP_ENTER: Synchronous GuestOS exception processing started.
- TRAP_LEAVE: Synchronous GuestOS exception processing finished.
- HYP_CALL: A GuestOS is using a hypercall on the current virtual CPU.
- VCPU_TO_EL3: Execution of a SMC instruction on behalf of a GuestOS started.
- VCPU_FROM_EL3: Execution of a SMC instruction on behalf of a GuestOS finished.
- VCPU_START: The current virtual CPU is going online.
- VCPU_STOP: The current virtual CPU is going offline.
- VCPU_SUSPEND_REQUEST: The specified virtual CPU has been requested to suspend.
- VCPU_RESUME_REQUEST: The specified virtual CPU has been requested to resume.
- VDEV_LOAD: Emulation of a load instruction from a virtual device.
- VDEV_STORE: Emulation of a store instruction to a virtual device.
- VIRQ_ASSERT: An interrupt is being asserted on the current virtual CPU.
- VCPU_SWITCH_OUT: The current virtual CPU is being de-scheduled from the current physical CPU.
- VCPU_SWITCH_IN: The current virtual CPU is being scheduled on the current physical CPU.
- VCPU_IDLE_ENTER: The current virtual CPU is entering idle state.
- VCPU_IDLE_LEAVE - The current virtual CPU is leaving idle state.

- VM_STOP: The current VM has been stopped.
- VM_START: The current VM is starting.
- VM_SUSPEND_REQUEST: The specified VM has been requested to suspend.
- VM_RESUME_REQUEST: The specified VM has been requested to resume.
- VM_LOADING: A GuestOS image loading is being started.
- VM_LOADED: A GuestOS image has been loaded.
- VM_FATAL: The current VM has triggered a fatal fault.

All payloads excepted VDEV_LOAD and VDEV_STORE have the following contents:

- ELR: Exception Link Address. The PC at which the exception occurred.
- ESR: Exception Syndrome Register. The reason of the event.
- SPSR: Saved Program Status register.

The vl,evt-log0/cpu-[id] file provides an ascii output formatted as follows:

```
[0] 457698172857: CPU0 VM2 VCPU0 - trap_leave
    pc:   0xffffff800876cac4
    spsr:         0x60400145
    esr:          0x5a00ff00
[1] 457698172877: CPU0 VM2 VCPU0 - irq_post
    target: VM2, VCPU0
    irq: 673
...
[3] 457698173025: CPU0 VM2 VCPU0 - irq_select
    irq: 0
[4] 457698173074: CPU0 VM2 VCPU0 - virq_assert
    physical IRQ id: 673
    virtual IRQ id: 513
...
[8] 457698174830: CPU0 VM2 VCPU0 - vdev_store
    vdev ID: 1
    access size: 4 bytes
    addr: 0x0000000010101f00
    data: 0x0000000000020000
```

Each record starts with a sequence number followed by a timestamp. Then the physical CPU, VM identifier and VCPU identifiers are provided. The event type is then provided. The following lines depend upon the event type.

By default, the Hypervisor does not monitor these events. To activate the monitoring of these events, one must perform the following command:

```
# echo echo -n 2 > /sys/nk/prop/nk.monitoring.enable
```

1.3.2.2.5.2 /proc/nk/monitoring/vl,evt-log-0-raw/cpu-[id]-records

This is a readable and mappable file providing access to the same information as */proc/nk/monitoring/vl,evt-log-0/cpu-[id]* file. However the information is provided in binary form.

The format of the records is defined in *nkevent-log.h*.

1.3.2.2.5.3 /proc/nk/monitoring/vl,evt-stats-0

This is a directory providing various files with event statistics matrix. Each matrix is provided as a file.

The occurrence of events is accounted in global counters. These counters are organized in a set of matrix.

There are global matrix, per-vm matrix and per-vcpu EL2 residence time frequency distribution matrix.

There are two global matrix:

- one for hardware interrupts
- and one for physical CPU operations.

In the global matrix, there is one row per physical CPU and one columns per event (hardware interrupt or CPU operation).

There are five matrix for each VM:

- one for virtual interrupts
- one for trap and synchronous exceptions
- one for hypercalls
- one for virtual device emulation (load and store operations)
- and one for VCPU operations

In the per-vm global matrix, there is one row per virtual CPU and one columns per event (interrupt, trap, hyper-call, ...).

There are up to nine EL2 residence time frequency distribution matrix for each VCPU:

- for hardware interrupt request EL2 execution time
- for hardware interrupt request EL2 processing wait time
- for hardware interrupt request EL2 virtualization latency
- for hyper-call EL2 execution time
- for hyper-call EL2 processing wait time
- for synchronous exception EL2 execution time
- for synchronous exception EL2 processing wait time
- for emulated memory access instruction EL2 execution time
- for emulated memory access instruction EL2 processing wait time

In frequency distribution matrix, there is one row per time interval (also called bucket) and one column per event (interrupt, trap, hyper-call, ...). Thus, for an event and a time interval the matrix cell counts the number of the event occurrences that EL2 processing time was contained within the time interval.

The actual number of frequency distribution matrices for a given VCPU, the number of buckets and the bucket's time intervals are configured by the Hypervisor Device Tree "vl,evt-stats" compatible node.

Internally, these matrix are organized as depicted. However, when displayed in ASCII though the following */proc/nk/monitoring* files, columns are printed as rows and rows as columns.

1.3.2.2.5.4 /proc/nk/monitoring/vl,evt-stats-0/hwirq-cntrs

This is the global matrix for hardware interrupts counters. The layout is as follows:

HWIRQ	IRQ0	IRQ1	...	IRQ max
CPU0	counter	counter	counter	counter
CPU1	counter	counter	counter	counter

There is one column for each hardware interrupt. Each cell counts the number of occurrences of a given hardware interrupt on a physical CPU.

```
        |            CPU0 |            CPU1 |            CPU2 |            CPU3 |            CPU4 |
IRQ-  0 |           86886 |          355113 |          355226 |          355126 |          355343 |
IRQ-  6 |               0 |               4 |               4 |               4 |               4 |
IRQ- 26 |            2028 |             601 |               0 |               0 |               0 |
IRQ- 27 |          613717 |          130753 |          590130 |            6361 |          102304 |
...
TOTAL   |         1337632 |          486562 |          945412 |          361600 |          457739 |
```

1.3.2.2.5.5 /proc/nk/monitoring/vl,evt-stats-0/cpu-ops-cntrs

This is the global matrix for operations on physical CPU counters. The layout is as follows:

CPUOPS	START	STOP	INTR_EL2	TO_EL3	IDLE	EXC_EL2	SPURIOUS_IRQ
CPU0	counter	counter	counter	counter	counter	counter	counter
CPU1	counter	counter	counter	counter	counter	counter	counter

For a definition of the CPU operations refer to the list of event types in /proc/nk/monitoring/vl,evt-log-0/cpu-[id].

```
              |            CPU0 |            CPU1 |            CPU2 |            CPU3 |            CPU4 |
CPU_START     |               0 |               1 |               1 |               1 |               1 |
INTR_EL2      |          959740 |          363477 |          946634 |          357363 |          393591 |
IDLE          |          888822 |          363118 |          944929 |          355825 |          393958 |
TOTAL         |         1848562 |          726596 |         1891564 |          713189 |          787550 |
```

1.3.2.2.5.6 /proc/nk/monitoring/vl,evt-stats-0/virq-[vmid]-cntrs

This a per-vm matrix for virtual interrupts. The layout is as follows:

VIRQ	VIRQ0	VIRQ1	...	VIRQ max
VCPU0	counter	counter	counter	counter
VCPU1	counter	counter	counter	counter

There is one column for each virtual interrupt. Each cell counts the number of occurrences of a given virtual interrupt on a virtual CPU.

```
          |           VCPU0 |           VCPU1 |           VCPU2 |           VCPU3 |           TOTAL |
VIRQ-  0  |           82693 |          179256 |          169014 |          170571 |          601534 |
VIRQ-  1  |               9 |             412 |            1297 |             948 |            2666 |
VIRQ-  5  |               1 |               0 |               0 |               0 |               1 |
VIRQ- 27  |          624222 |          131747 |          106598 |           41217 |          903784 |
...
TOTAL     |         1345542 |          311418 |          276931 |          212793 |         2146684 |
```

1.3.2.2.5.7 /proc/nk/monitoring/vl,evt-stats-0/trap-[vmid]-cntrs

This a per-vm matrix for synchronous exceptions. The layout is as follows:

TRAP	ESC0	ESC1	...	ESC max
VCPU0	counter	counter	counter	counter
VCPU1	counter	counter	counter	counter

There is one column for each exception code. Each cell counts the number of occurrences of a given exception on a virtual CPU.

```
                                |          VCPU0  |          VCPU1  |          VCPU2  |          VCPU3  |
ESR.EC- 1 'WFI/WFE'             |         899852  |         365929  |         401784  |         194510  |
ESR.EC-22 'HVC AArch64'         |              2  |              0  |              0  |              1  |
ESR.EC-23 'SMC AArch64'         |             10  |              2  |              3  |              2  |
ESR.EC-24 'MSR/MRS'             |             21  |             21  |             21  |             21  |
ESR.EC-36 'DAbort EL0/1'        |         508659  |         206757  |          98359  |         139997  |
TOTAL                           |        1408544  |         572709  |         500167  |         334531  |
```

1.3.2.2.5.8 /proc/nk/monitoring/vl,evt-stats-0/hcall-[vmid]-cntrs

This a per-vm matrix for hypercall invocations. The layout is as follows:

HCALL	HCALL0	HCALL1	...	HCALL 127
VCPU0	counter	counter	counter	counter
VCPU1	counter	counter	counter	counter

There is one column for each hypercall. Each cell counts the number of occurrences of a given hypercall on a virtual CPU.

```
                                |          VCPU0  |          VCPU1  |          VCPU2  |          VCPU3  |
HCALL- 6 'timer_getfreq'        |              0  |              0  |              0  |              1  |
HCALL- 8 'last_vm_id'           |              0  |              0  |              0  |              1  |
HCALL- 9 'my_vm_id'             |             10  |             15  |             74  |             29  |
...
TOTAL                           |           1206  |            647  |          31731  |           2567  |
```

1.3.2.2.5.9 /proc/nk/monitoring/vl,evt-stats-0/vcpu-ops-[vmid]-cntrs

This a per-vm matrix for VCPU operations. The layout is as follows:

VCPUOPS	START	STOP	SWITCH	TO_EL3	IDLE	VMSTART	VMSTOP	VMLOADING	VM_FATAL	MIGRATE
VCPU0	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter
VCPU1	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter

There is one column for each kind of VCPU operation. Each cell counts the number of occurrences of a given operation on a virtual CPU. For a definition of the VCPU operations refer to the list of event types in /proc/nk/monitoring/vl,evt-log-0/cpu-[id].

```
                 |          VCPU0  |          VCPU1  |          VCPU2  |          VCPU3  |          TOT
VCPU_START       |              1  |              1  |              1  |              1  |
TO_EL3           |              7  |              2  |              3  |              2  |
```

```
IDLE                    |        903325 |        364704 |        399886 |        189947 |        18578
VM_START                |             1 |             0 |             0 |             0 |
TOTAL                   |        903334 |        364707 |        399890 |        189950 |        18578
```

1.3.2.2.5.10 /proc/nk/monitoring/vl,evt-stats-0/vdev-acc-[vmid]-cntrs

This a per-vm matrix for virtual devices access. The layout is as follows:

VDEV	<vdev-0>-r@<addr-0>	<vdev-0>-w@<addr-0>	<vdev-1>-r@<addr-1>	<vdev-1>-w@<addr-1>	
VCPU0	counter	counter	counter	counter	counter
VCPU1	counter	counter	counter	counter	counter

There are two columns per virtual device. One to count the load accesses (*-r@*) and one to count the store accesses (*-w@*). The *addr-i* field is the base address of the virtual device within the virtual machine. It is an Intermediate Physical Address (IPA).

```
                              |          VCPU0 |          VCPU1 |          VCPU2 |          VCPU3 |
arm,gic-v2/gicd-r@10101000 |             28 |             21 |             56 |             65 |
arm,gic-v2/gicd-w@10101000 |         507922 |         154692 |          66057 |          71890 |
VS5PV210 UART-r@10300000   |            356 |          27180 |            729 |          36017 |
VS5PV210 UART-w@10300000   |            349 |          24659 |            707 |          30723 |
vGPIO-r@10230000           |              0 |              4 |              0 |             16 |
vGPIO-w@10230000           |              0 |              8 |              0 |             16 |
...
TOTAL                      |         508682 |         206612 |          67608 |         138823 |
```

1.3.2.2.5.11 /proc/nk/monitoring/vl,evt-stats-0/el2time/hwirq-exec-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for hardware interrupt request execution time. The layout is as follows:

	IRQ0	IRQ1	...	IRQ max
bucket-0	counter	counter	counter	counter
bucket-1	counter	counter	counter	counter
...	...	...	...	...

There is one column for each hardware interrupt. For a given VCPU and a given hardware interrupt each cell counts the number of occurrences of the interrupt requests that EL2 execution time was contained within a bucket.

```
        |         <2.4us |         <4.9us |         <9.8us |        >=9.8us |        AVERAGE |
IRQ-  0 |           2420 |           1372 |              5 |              3 |          2.5us |
IRQ- 15 |             63 |              1 |              0 |              0 |          1.8us |
IRQ- 26 |              0 |              0 |              0 |              1 |           14us |
IRQ- 27 |             96 |           1323 |            331 |              1 |          4.2us |
IRQ-223 |             10 |             30 |              0 |              0 |          3.2us |
IRQ-260 |           2088 |           5690 |            324 |              0 |          3.3us |
IRQ-268 |             55 |             13 |              0 |              0 |          2.1us |
IRQ-275 |             27 |            109 |             22 |              0 |          3.8us |
IRQ-287 |           1421 |            499 |             34 |              0 |          2.3us |
IRQ-288 |            235 |            391 |              3 |              0 |          3.0us |
IRQ-289 |              4 |              3 |              1 |              0 |          3.2us |
IRQ-391 |              1 |              0 |              0 |              0 |          1.8us |
TOTAL   |           6420 |           9431 |            720 |              5 |          3.1us |
```

1.3.2.2.5.12 /proc/nk/monitoring/vl,evt-stats-0/el2time/hwirq-wait-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for hardware interrupt request EL2 processing wait time - the time a VCPU was waiting (not running) while processing the interrupt request.

A VCPU is considered as waiting when it's preempted, typically at the end of the interrupt processing, by another VCPU.

The layout of this matrix is the same as /proc/nk/monitoring/vl,evt-stats-0/el2time/hwirq-exec-[vmid]-[vcpuid]-cntrs.

1.3.2.2.5.13 /proc/nk/monitoring/vl,evt-stats-0/el2time/hwirq-latency-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for hardware interrupt request EL2 virtualization latency. The layout is as follows:

	VIRQ0	VIRQ1	...	VIRQ max
bucket-0	counter	counter	counter	counter
bucket-1	counter	counter	counter	counter
...	...	...	...	...

There is one column for each virtual interrupt. For a given VCPU and a given virtual interrupt each cell counts the number of occurrences of the corresponding physical interrupt requests that EL2 virtualization latency was contained within a bucket.

A hardware interrupt virtualization latency is the elapsed time form the moment when HV takes a physical hardware interrupt (asynchronous exception) at EL2 until the moment when a VCPU returns at EL1 with the corresponding virtual hardware interrupt asserted in the VCPU's GICH (LR register).

```
          |           <2.4us |           <4.9us |           <9.8us |          >=9.8us |          AVERAGE |
VIRQ- 54  |                0 |                1 |                0 |                0 |            3.6us |
VIRQ- 56  |                0 |                1 |                1 |                0 |            5.5us |
VIRQ- 81  |                0 |                1 |                0 |                0 |            3.6us |
VIRQ-167  |              416 |              960 |              812 |                3 |            4.6us |
VIRQ-168  |             1285 |              865 |               42 |                0 |            2.6us |
VIRQ-231  |               16 |               39 |                3 |                0 |            3.3us |
VIRQ-255  |              123 |               13 |                0 |                0 |            2.0us |
VIRQ-259  |              542 |             2494 |               28 |                0 |            3.3us |
VIRQ-267  |              117 |              305 |                4 |                0 |            3.1us |
VIRQ-279  |              430 |              133 |                0 |                0 |            2.2us |
VIRQ-280  |            35135 |           147407 |              980 |                8 |            3.3us |
VIRQ-281  |                1 |                2 |                0 |                0 |            3.0us |
VIRQ-283  |                0 |                1 |                0 |                0 |            3.6us |
VIRQ-295  |              568 |             1590 |               32 |                1 |            3.2us |
VIRQ-298  |               15 |               97 |                2 |                0 |            3.5us |
TOTAL     |            38648 |           153909 |             1904 |               12 |            3.3us |
```

1.3.2.2.5.14 /proc/nk/monitoring/vl,evt-stats-0/el2time/hwirq-vswitch-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for hardware interrupt request EL2 virtualization latency involving virtual CPU switch. The layout is as follows:

	VIRQ0	VIRQ1	...	VIRQ max
bucket-0	counter	counter	counter	counter
bucket-1	counter	counter	counter	counter
...	...	...	...	...

There is one column for each virtual interrupt. For a given VCPU and a given virtual interrupt each cell counts the number of occurrences of the corresponding physical interrupt requests that EL2 virtualization latency was contained within a bucket.

This matrix captures the virtualization latency of the interrupt requests taken in the context of one VCPU but asserted in GICH (LR register) of another VCPU. In other words the scope of this matrix is the interrupt requests whose virtualization involves a VCPU switch.

1.3.2.2.5.15 /proc/nk/monitoring/vl,evt-stats-0/el2time/hcall-exec-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for hypr-call EL2 execution time. The layout is as follows:

	HCALL0	HCALL1	...	HCALL max
bucket-0	counter	counter	counter	counter
bucket-1	counter	counter	counter	counter
...	...	...	...	...

There is one column for each hyper-call. For a given VCPU and a given hypr-call each cell counts the number of occurrences of the hyper-call requests that EL2 execution time was contained within a bucket.

```
                                 |          <2.4us |          <4.9us |          <9.8us |          >=9.8us |
HCALL- 9 'my_vm_id'              |             291 |               7 |               0 |                0 |
HCALL-11 'vlink_lookup'          |             191 |               1 |               0 |                0 |
HCALL-13 'pmem_alloc'            |              15 |               4 |               7 |               24 |
HCALL-14 'pxirq_alloc'           |              48 |               0 |               0 |                0 |
HCALL-15 'xirq_ptov'             |              48 |               0 |               0 |                0 |
HCALL-16 'xirq_post'             |             228 |             800 |               7 |               24 |
HCALL-22 'prop_get'              |            1170 |             234 |              64 |                2 |
HCALL-57 'get_cpuid'             |               2 |               0 |               0 |                0 |
HCALL-72 'vm_mem_verify'         |              97 |               3 |               0 |                0 |
HCALL-73 'vm_mem_verify_and_loc  |               1 |              58 |              83 |                7 |
HCALL-74 'vm_mem_unlock'         |               0 |               0 |             101 |                2 |
HCALL-97                         |               7 |               0 |               3 |                9 |
TOTAL                            |            2098 |            1107 |             265 |               68 |
```

1.3.2.2.5.16 /proc/nk/monitoring/vl,evt-stats-0/el2time/hcall-wait-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for hyper-call request EL2 processing wait time - the time a VCPU was waiting (not running) while processing the hyper-call.

A VCPU is considered as waiting when it's preempted by another VCPU

The layout of this matrix is the same as /proc/nk/monitoring/vl,evt-stats-0/el2time/hcall-exec-[vmid]-[vcpuid]-cntrs.

1.3.2.2.5.17 /proc/nk/monitoring/vl,evt-stats-0/el2time/trap-exec-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for synchronous exception EL2 execution time. The layout is as follows:

	ESR.EC0	ESR.EC1	...	ESR.EC max
bucket-0	counter	counter	counter	counter
bucket-1	counter	counter	counter	counter
...	...	...	...	...

There is one column for each Exception Code as reported by EC field of ESR_EL2 register. For a given VCPU and a given EC each cell counts the number of occurrences of the exceptions that EL2 execution time was contained within a bucket.

```
                                 |          <2.4us |          <4.9us |          <9.8us |          >=9.8us |
ESR.EC- 0                        |               0 |               0 |               0 |                1 |
```

```
ESR.EC- 1 'WFI/WFE'              |        313448 |          2345 |             5 |             7 |
ESR.EC-23 'SMC AArch64'          |            57 |            10 |             0 |             2 |
ESR.EC-24 'MSR/MRS'              |            20 |             0 |             2 |             0 |
TOTAL                            |        313525 |          2355 |             7 |            10 |
```

1.3.2.2.5.18 /proc/nk/monitoring/vl,evt-stats-0/el2time/trap-wait-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for synchronous exception EL2 processing wait time - the time a VCPU was waiting (not running) while processing the exception.

A VCPU is considered as waiting when:

- preempted on a physical CPU by another VCPU, or
- running in the secure world after performing an SMC

The layout of this matrix is the same as /proc/nk/monitoring/vl,evt-stats-0/el2time/trap-exec-[vmid]-[vcpuid]-cntrs.

1.3.2.2.5.19 /proc/nk/monitoring/vl,evt-stats-0/el2time/vdev-acc-exec-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for EL2 execution time of a stage-2 data abort synchronous exception made by a memory access instruction.

The layout is as follows:

VDEV	<vdev-0>-r@<addr-0>	<vdev-0>-w@<addr-0>	<vdev-1>-r@<addr-1>	<vdev-1>-w@<addr-1>	
bucket-0	counter	counter	counter	counter	counter
bucket-1	counter	counter	counter	counter	counter
...	...	...	...	...	...

There are two columns per virtual device. One to count the load accesses (*-r@*) and one to count the store accesses (*-w@*). The *addr-i* field is the base address of the virtual device within the virtual machine. It is an Intermediate Physical Address (IPA).

```
                               |        <2.4us |        <4.9us |        <9.8us |       >=9.8us |
arm,gic-v2/gicd-r@10101000     |           344 |             8 |             6 |             0 |
arm,gic-v2/gicd-w@10101000     |          7863 |         17783 |          1762 |            29 |
vDINTC-r@10030000              |            71 |            62 |             7 |             0 |
vDINTC-w@10030000              |           423 |           399 |            49 |             1 |
VS5PV210 UART-r@10300000       |            24 |             1 |             0 |             0 |
VS5PV210 UART-w@10300000       |            34 |             1 |             1 |             0 |
vGPIO-w@10450000               |             0 |             0 |             2 |             1 |
vgpio-eint-w@10230900          |             6 |             0 |             2 |             0 |
vGPIO-w@10230000               |             3 |             3 |             6 |             0 |
vgpio-eint-w@10830900          |             7 |             2 |             1 |             2 |
vGPIO-r@10830000               |            18 |             7 |             1 |             0 |
vGPIO-w@10830000               |            43 |             0 |             0 |             0 |
vgpio-eint-w@17740900          |             3 |             0 |             1 |             0 |
vGPIO-w@17740000               |             2 |             1 |             3 |             0 |
vgpio-eint-w@17060900          |             2 |             0 |             0 |             0 |
vGPIO-w@17060000               |             0 |             0 |             3 |             0 |
vgpio-eint-w@17c30900          |             8 |             0 |             2 |             0 |
vGPIO-w@17c30000               |             6 |             1 |             7 |             0 |
IOMEM-r@10460708               |             1 |             2 |             0 |             0 |
IOMEM-w@10460708               |             3 |             0 |             0 |             0 |
...                            |           ... |           ... |           ... |           ... |
TOTAL                          |          8879 |         18273 |          1854 |            33 |
```

1.3.2.2.5.20 /proc/nk/monitoring/vl,evt-stats-0/el2time/vdev-acc-wait-[vmid]-[vcpuid]-cntrs

This is the frequency distribution matrix for EL2 processing wait time of a stage-2 data abort synchronous exception made by a memory access instruction. Wait time is the time a VCPU was waiting (not running).

A VCPU is considered as waiting when:

- preempted on a physical CPU by another VCPU, or
- blocked in Hypervisor waiting for completion of an action processed by another VCPU

The layout of this matrix is the same as /proc/nk/monitoring/vl,evt-stats-0/el2time/vdev-acc-exec-[vmid]-[vcpuid]-cntrs.

1.3.2.2.5.21 /proc/nk/pcpustats

This provides a matrix of time spent in the various virtual machines for each physical CPU

CPUSTATS	VM1	VM2	VM3	...
CPU 0	time	time	time	time
CPU 1	time	time	time	time

There is one row per physical CPU and one column per virtual machine. VM1 is actually the Hypervisor. The time spent in virtual machines accounts the time spent running at EL0 and EL1. Time is accounted in Hypervisor ticks.

```
Counting frequency: 26000000 Hz
CPU         Idle          VM1          VM2          VM3
  0    395573346 569979943344       228876            0
  1    395836376 569979883340        25912            0
  2    395679564 569979933609            0       132493
  3    395866329 569979878131            0         1250
  4    395775378 569979897703        72665            0
  5    395839529 569979878176        28074            0
  6    395866404 569979877036            0         2370
  7    395865686 569979877277            0         2880
In msecs:
  0        15214     21922305            8            0
  1        15224     21922303            0            0
  2        15218     21922305            0            5
  3        15225     21922303            0            0
  4        15222     21922303            2            0
  5        15224     21922303            1            0
  6        15225     21922302            0            0
  7        15225     21922302            0            0
```

In order to activate cpu statistics you have to first issue the following command:

```
echo -n 1 > /sys/nk/prop/nk.monitoring.enable
```

1.3.2.2.5.22 /proc/nk/props

This read-only file provides an abstract of the properties defined on the system.

```
PID NAME                  ATTR LEN VALUE
0   nk.error.ignore-alloc r--    4 31
1   nk.error.ignore-non-a r--    4 31
2   nk.vm.2.bootcomplete  rwn    4 Not set
3   nk.vm.3.bootcomplete  r-n    4 Not set
4   nk.cpus.online        r-n    4 ff
```

The following fields are lister for each listed property:

- *PID*: identifier of the property
- *NAME*: name of the property. The name might be truncated if it is longer than 21 characters.
- *ATTR*: attributes of the property. This determines whether the current Guest OS can read, write the property, or be notified upon property value modification.
- *LEN*: the length in bytes of the value of the property.
- *VALUE*: An hexadecimal dump of the first bytes of the value.

Properties can be accessed by their full name through the */sys/nk/props/[property_name]* file.

1.3.2.2.5.23 /proc/nk/regions

This read-only file provides information about the guest images that can be loadied dynamically. When reading this file, it shows the base address and size of each VM kernel and initrd regions encoded in the nk.vm.∗.kernel.region and nk.vm.∗.initrd.region Hypervisor properties.

In a configuration which doesn't require a dynamic guest image loading, these properties are typically omitted. So, this file only shows a banner string. Otherwise, it shows the loadable images physical address and size (see example below).

```
# cat /proc/nk/regions
PADDR       SIZE        ID
0xA0000000  0x01000000  vm.3.kernel
0xA2000000  0x00200000  vm.3.initrd
```

1.3.2.2.5.24 /proc/nk/restart

Writing a Guest OS id in this file triggers a restart of that Guest OS. Equivalent to:

- echo [id] > /proc/nk/vmstop
- echo [id] > /proc/nk/vmstart

The only difference is that

- echo [id] > /proc/nk/restart works for any VM including the current one.

This is equivalent to writing values "0" and then "1" into the */sys/nk/props/nk.vm.[id].state.running* file. See the "↩ Hypervisor Reference manual > Programming Interface for Guest OS > Hypervisor properties > VM Management" documentation.

In order to restart the current Guest OS, it is also possible to perform:

- echo [id] > /proc/nk/vmstart

1.3.2.2.5.25 /proc/nk/state

Reading this read-only file returns a bit mask of the running or stop state for each Guest OS. The bit mask is printed as a decimal string. A bit set to 1 means the corresponding Guest OS is running. A bit set to 0 means the corresponding Guest OS is stopped.

The string is not null terminated.

```
# cat /proc/nk/state
12
```

In the above example, the returned value is 12, which converts to hexadecimal 0xC. This means that Guest OSes 2 and 3 are running. OS 0 is the Hypervisor and OS 1 is unused.

1.3.2.2.5.26 /proc/nk/vmresume

Writing a Guest OS id in this file triggers a resume of that Guest OS as long as the Guest OS is in the "paused" state. Resuming a Guest OS which is not paused has no effect. This is equivalent to writing the value "0" into the */sys/nk/props/nk.vm.[id].state.paused* file. See the "Hypervisor Reference manual > Programming Interface for Guest OS > Hypervisor properties > VM Management" documentation.

1.3.2.2.5.27 /proc/nk/vmstart

Writing a GuestOS id different from the current one triggers a boot (start) of that Guest OS as long as the target Guest OS is in the "stopped" state. To stop a Guest OS one may use the */proc/nk/stop* file or the */proc/nk/vmstop* file. Starting a running Guest OS has no effect.

Writing the current GuestOS id triggers restart of the current GuestOS. This is equivalent to: echo [id] > /proc/nk/restart

This is equivalent to writing the value "1" into the */sys/nk/props/nk.vm.[id].state.running* file. See the "Hypervisor Reference manual > Programming Interface for Guest OS > Hypervisor properties > VM Management" documentation.

1.3.2.2.5.28 /proc/nk/vmstop

Writing a Guest OS id in this file triggers a stop of that Guest OS as long as the target Guest OS is the "running" state. Stopping a stopped Guest OS has no effect. This is equivalent to writing the value "0" into the */sys/nk/props/nk.vm.[id].state.running* file.

1.3.2.2.5.29 /proc/nk/vmsuspend

Writing a Guest OS id in this file triggers a suspend of that Guest OS as long as the target Guest OS is in the "playing" state. Suspending a "paused" Guest OS has no effect.

This is equivalent to writing the value "0" into the */sys/nk/props/nk.vm.[id].state.running* file.

1.3.2.2.5.30 /proc/nk/xirqmap

This read-only file provides the mapping between interrupts in the VM space and interrupts in the space of the VM virtual interrupt controller (vGIC).

```
  0 -   15 =>    invalid (16)
 16 -   26 =>   16 -   26 (11)
 27 -   27 =>    invalid (1)
```

There is one line per range of interrupts in VM space which are mapped contiguously to interrupts in the virtual interrupts controller space as seen by the GuestOS interrupt controller driver

The first interval is the interval of virtual interrupts as seen by the Guest OS. The second interval is the physical interrupts interval as seen by the Hypervisor. The number in parenthesis is the number of interrupts within the interval.

1.3.2.2.5.31 /proc/nk/counter-timer-offset

This read-only file provides, expressed in hypervisor clock tick unit, the offset of the hypervisor clock from 'POSIX.1 Epoch, 1970-01-01 00:00:00 +0000 (UTC)' date.

This offset allows for timestamps expressed in hypervisor clock tick count to be reported as epoch dates instead of clock ticks since system startup time.

Used hypervisor timestamping is based on hyp_call_timer_getcount() which reports the number of a reference clock ticks elapsed since the start of the system. The frequency of the the related clock is available through the hyp_call_timer_getfreq() function.

The offset is computed in term of the number of ticks of a clock running at the frequency reported by hyp_call_timer_getfreq().

The offset is dependent on the value of the guest OS system date, if the reference from which the system date is set changes, then the offset reported by the /proc/nk/counter-timer-offset file will also change.

As hypervisor clock has a frequency dependending on the target hardware and as the system date is expressed in nanoseconds, the offset is computed using average between two hypervisor clock value readings separated by the reading of the system current date.

This average value is computed at each /proc/nk/counter-timer-offset and can vary slightly from one read to another, depending on the actua scheduling of the VM guest OS and hypervisor implieds calls.

1.3.2.3 NOTES

1.3.2.4 SEE ALSO

/sys/nk(5)

Hypervisor Event Logging in "Device Virtualization reference manual".

1.3.3 <debugfs>/vlx-cpu-hotplug_debugfs/no-action-on-cpus(5)

1.3.3.1 Cross References

Related Documents
Manual Page

1.3.3.2 NAME

vlx-cpu-hotplug/no-action-on-cpus — Tuning and debugging API for CPU handover vdriver

1.3.3.3 DESCRIPTION

Purpose of the tuning is to allow for driver to be inhibited and not to process CPU operation related to handover requests.

Purpose of the inhibited mode is to allow tests over not processed CPU handover requests. Tester can verify that requests which are not properly carried out are detected. Developer can test that their fallback policy and corrective actions are effectively brought into play.

1.3.3.3.1 Configuration

If CPU handover debugfs API must be used then it is necessary to activate it as an option of the built-in driver for the VM kernel.

```
CONFIG_VLX_CPU_HOTPLUG_DEBUG=y
```

1.3.3.3.2 Files and Directories

1.3.3.3.2.1 vlx-cpu-hotplug/no-action-on-cpus

```
The file '/sys/kernel/debug/vlx-cpu-hotplug/no-action-on-cpus' is
created by the vlx-cpu-hotplug vdriver when compiled with the
VLX_CPU_HOTPLUG_DEBUG configuration option.
It instanciates a debugfs u32 entry which can take the
following values:
    '0' (default): the driver behaves accordingly to its
                   specifications. CPU handover requests are
                   processed and CPU related operations are
                   made.

    '1': the driver is inhibited and will not process CPU
             handover requests.

Purpose of the inhibited mode is to allow tests over not processed
CPU handover requests. Tester can verify that requests which are
not properly carried out are detected. Developer can test that
their fallback policy and corrective actions are effectively
brought into play.
```

1.4 (8) Administration and privileged commands

vgki-helper(8)

1.4.1 vgki-helper(8)

1.4.1.1 Cross References

Related Documents
Component Interface
Manual Page

1.4.1.2 NAME

vgki-helper — User-mode helper process for `vgki` driver

1.4.1.3 DESCRIPTION

The `vgki-helper` process is a daemon which actually performs the bulk of processing requests from the VGKI kernel-mode API. It must be running for the VGKI KAPI requests to return. The `vgki-helper` process should normally be started during the system boot. It will immediately switch to background. It needs to run with privileges high enough to open the files and devices requested through vgki_kapi_filp_open() and vgki_kapi_open() calls, typically as user root. Attempting to run a second instance will fail until the first one terminates and closes the `vgki` driver.

1.4.1.4 CONFIGURATION

The `vgki-helper` process does not need command-line options during normal operations. It offers the following debug-oriented ones:

Option	Purpose
-e *filename*	Redirect standard error output to *filename*, a device or a file
-f	Run in foreground instead of switching to background. Signal `SIGINT` can be used to perform a clean shutdown if the `vgki-helper` process is idle
-h	Print help and exit
-t *thread_count*	Change the allowed maximum number of server threads. The default is 128
-v	Run in verbose mode

1.4.1.5 FILES AND DIRECTORIES

The `vgki-helper` process accesses the `/dev/vgki` device file managed by the `vgki` driver, and files requested through the VGKI kernel-mode API.

It establishes monitoring of the `/dev` directory to detect the loading of the `vgki` driver just in time. This monitoring is not required for operations.

1.4.1.6 OBSERVATION

The `vgki-helper` process can be sent the *SIGUSR1* signal, and will dump internal state to standard error output. Only non-zero counters are displayed. For example:

```
vgki_sigusr1_handler(28870): Got SIGUSR1 (10), dumping state
vgki_dump(28870): idle 3, running 3, fd 4, main_tid 28870, aborting 0, verbose 0, max_threads 128
vgki_dump(28870): Stats: pthread_create:8006 thread_return:8004 in-NONE:11007 in-FILP_OPEN:1000 in-KERNEL_THRE
```

Note

> When started as daemon on Android by `/init`, the `vgki-helper` process has `/dev/null` as standard error output, so no traces are visible on console.

In general, it is not possible to terminate the `vgki-helper` process when actively used, because this means all VGKI threads must exit. When the `vgki-helper` process is idle, it is possible to terminate it cleanly by sending the *SIGINT* signal (internal state will be dumped as above), or brutally, with any other signal.

Chapter 2

Test List

Global _htf_vgic_vdev_run (htf_gtest_id_t tid, void ∗cookie, char ∗args, size_t size)

This test case provides

Chapter 3

Deprecated List

Page /sys/nk(5)

/sys/nk/prop/nk.cpustats.hyp.enable is deprecated. See /sys/nk/prop/nk.monitoring.enable instead.

Chapter 4

Module Index

4.1 Driver API

Here is the list of API description documents

Chapter 5

Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Module Documentation

7.1 Virtual MBOX API

Data Structures

- struct vmbox_cb_ops_t

  Structure containing call-back notifications. More...

Typedefs

- typedef void(∗ vmbox_receive_t) (void ∗cookie, unsigned int vmid, void ∗vmq, void ∗msg)

  Define the type of the notification call back which is invoked each time a new message is received. The ***vmid*** *argument identifies the sender VM. The* ***vmq*** *argument identifies the reception queue. The queue identifier should be used to release the incoming message by a subsequent vmbox_msg_free invocation. The* ***msg*** *argument points to the message body.*
- typedef void(∗ vmbox_link_state_t) (void ∗cookie, unsigned int vmid)

  Type defining notification call-backs which are invoked when the virtual link to a given VM becomes online or offline.

Functions

- int vmbox_init (const char ∗name, bool frontend, unsigned int msgsize, unsigned int msgcount, void ∗∗phandle)

  Start working with the virtual Message Box library.
- void vmbox_cb_ops_register (void ∗handle, vmbox_cb_ops_t ∗ops, void ∗cookie)

  Register call back operations.
- unsigned int vmbox_vms_mask (void ∗handle)

  Get a mask of connected VMs.
- int vmbox_msg_alloc (void ∗handle, unsigned int vmid, void ∗∗vmq, void ∗∗msg)

  Allocate a transmission message.
- int vmbox_msg_alloc_early (void ∗handle, unsigned int vmid, void ∗∗vmq, void ∗∗msg)

  Allocate a transmission message for early sending.
- int vmbox_msg_send (void ∗vmq, void ∗msg)

  Send message.
- int vmbox_msg_send_early (void ∗vmq, void ∗msg)

  Send message early.
- int vmbox_msg_free (void ∗vmq, void ∗msg)

  Free received message.
- void vmbox_exit (void ∗handle)

  Stop working with the virtual message box library.

Variables

- vmbox_receive_t vmbox_cb_ops_t::receive

  call back invoked when receiving messages.
- vmbox_link_state_t vmbox_cb_ops_t::on

  call back invoked when link goes online.
- vmbox_link_state_t vmbox_cb_ops_t::off

  call back invoked when link goes offline.

7.1.1 Description

7.1.2 Data Structure Documentation

7.1.2.1 struct vmbox_cb_ops_t

Structure containing call-back notifications.

Definition at line 35 of file vmbox.h.

Data Fields

vmbox_link_state_t	on	call back invoked when link goes online.
vmbox_link_state_t	off	call back invoked when link goes offline.
vmbox_receive_t	receive	call back invoked when receiving messages.

7.1.3 Typedef Documentation

7.1.3.1 vmbox_receive_t

```
typedef void(* vmbox_receive_t)(void *cookie, unsigned int vmid, void *vmq, void *msg)
```

Define the type of the notification call back which is invoked each time a new message is received. The **vmid** argument identifies the sender VM. The **vmq** argument identifies the reception queue. The queue identifier should be used to release the incoming message by a subsequent vmbox_msg_free invocation. The **msg** argument points to the message body.

Parameters

cookie	the caller cookie given at call back registration time
vmid	the peer (remote) VM identifier
vmq	the queue identifier
msg	the message body

Definition at line 30 of file vmbox.h.

7.1.3.2 vmbox_link_state_t

```
typedef(* vmbox_link_state_t)(void *cookie, unsigned int vmid)
```

Type defining notification call-backs which are invoked when the virtual link to a given VM becomes online or offline.

Parameters

cookie	the caller cookie given at call back registration time
vmid	the peer (remote) VM identifier

Definition at line 33 of file vmbox.h.

7.1.4 Function Documentation

7.1.4.1 vmbox_init()

```
int vmbox_init (
            const char * name,
            bool frontend,
            unsigned int msgsize,
            unsigned int msgcount,
            void ** handle )
```

Start working with the virtual Message Box library.

The caller invokes this function first in order to start working with the library. This function searches for all existing virtual links matching a given **name** string. The caller specifies its role (back-end vs. front-end) by the **frontend** argument. Note that this flag is only used for a loop back communication when both sides are running in a same VM. On success, the function returns a zero value and stores a handle at address provided by the **handle** argument. A handle designates a message box instance and it should be passed back in all subsequent invocations.

The **msgsize** and **msgcount** arguments should have the same values on all communication sides.

Parameters

name	the **compatible** property string of the virtual link node
frontend	the Boolean value
msgsize	the message size
msgcount	the message queue size (must be a power of 2)
handle	the address where a message box handle is returned on success

Returns

0 - success
-EINVAL - **msgcount** is not a power of 2
-ENOMEM - no enough memory to create an instance descriptor

Definition at line 398 of file vmbox.c.

7.1.4.2 vmbox_cb_ops_register()

```
void vmbox_cb_ops_register (
            void * handle,
            vmbox_cb_ops_t * ops,
            void * cookie )
```

Register call back operations.

This function registers the caller call backs. The **cookie** argument is an opaque value for the library and it will just be passed back to the caller as a call back argument. This makes it possible for the caller to associate its own descriptor to the message box instance and then to get this descriptor available in the call back.

It is important to underline that all call back invocations for a given message box instance will be performed by a single thread.

Parameters

handle	the message box instance handle
ops	the structure defining the call backs
cookie	the caller descriptor associated with the message box instance

Definition at line 471 of file vmbox.c.

7.1.4.3 vmbox_vms_mask()

```
unsigned int vmbox_vms_mask (
            void * handle )
```

Get a mask of connected VMs.

This function returns a bit mask where each bit position corresponds to a VM identifier. Bit set means that the corresponding VM is connected through a virtual link.

Parameters

handle	the message box instance handle

Returns

bit mask of connected VMs

Definition at line 558 of file vmbox.c.

7.1.4.4 vmbox_msg_alloc()

```
int vmbox_msg_alloc (
            void * handle,
            unsigned int vmid,
            void ** vmq,
            void ** msg )
```

Allocate a transmission message.

This function allocates a transmission message for a given message box instance. The destination VM is identified by the **vmid** argument. A message queue handle is stored at the address specified by the **vmq** argument. A pointer to the message body is stored at the address specified by the **msg** argument. Once a message is successfully allocated and filled in with payload data, the caller should enqueue the message by invoking vmbox_msg_send function.

Note that currently there is no way to drop an allocated message without sending it.

Parameters

handle	the queue instance handle
vmid	the destination VM identifier
vmq	the address where a message queue handle is stored
msg	the address where a pointer to the allocated message body is stored

Returns

0 - success
-ENOTCONN - virtual link is offline
-EAGAIN - no more free messages (the transmission queue is full)
-EINVAL - queue meta data is corrupted

Definition at line 499 of file vmbox.c.

7.1.4.5 vmbox_msg_alloc_early()

```
int vmbox_msg_alloc_early (
            void * handle,
            unsigned int vmid,
            void ** vmq,
            void ** msg )
```

Allocate a transmission message for early sending.

This function is similar to the vmbox_msg_alloc one. The only difference is that the library allows to allocate a message when the virtual link is still offline but the local endpoint is already ready. It should be understood that the virtual link is online when both endpoints are ready for the communication. Thus, this function allows for an early message allocation while the remote endpoint is not ready yet.

Parameters

handle	the queue instance handle
vmid	the destination VM identifier
vmq	the address where a message queue handle is stored
msg	the address where a pointer to the allocated message body is stored

Returns

0 - success
-ENOTCONN - local virtual link endpoint is not ready
-EAGAIN - no more free messages (the transmission queue is full)
-EINVAL - queue meta data is corrupted

Definition at line 519 of file vmbox.c.

7.1.4.6 vmbox_msg_send()

```
int vmbox_msg_send (
            void * vmq,
            void * msg )
```

Send message.

This function puts a given message to a given transmission queue. The **vmq** argument points to the message queue handle returned by either vmbox_msg_alloc or vmbox_msg_alloc_early. The **msg** argument points to the message body previously allocated by either vmbox_msg_alloc or vmbox_msg_alloc_early.

Parameters

vmq	the message queue handle
msg	the message body address

Returns

0 - success
-ENOTCONN - virtual link is offline
-EINVAL - queue meta data is corrupted

Definition at line 540 of file vmbox.c.

7.1.4.7 vmbox_msg_send_early()

```
int vmbox_msg_send_early (
            void * vmq,
            void * msg )
```

Send message early.

This function is similar to the vmbox_msg_send one. The only difference is that the library allows to enqueue a message when the virtual link is still offline but the local endpoint is already ready. It should be understood that the virtual link is online only when both endpoints are ready for the communication. Thus, this function allows for an early message sending while the remote endpoint is not ready yet. This message will be received once the remote endpoint becomes ready.

Parameters

vmq	the message queue handle
msg	the message body address

Returns

0 - success
-ENOTCONN - local virtual link endpoint is not ready
-EINVAL - queue meta data is corrupted

Definition at line 546 of file vmbox.c.

7.1.4.8 vmbox_msg_free()

```
int vmbox_msg_free (
            void * vmq,
            void * msg )
```

Free received message.

This function releases a given message to a given reception queue. The **vmq** argument points to the message queue handle previously obtained by vmbox_cb_ops_t::receive call back. The **msg** argument points to the message body previously obtained by vmbox_cb_ops_t::receive call back.

Parameters

vmq	the message queue handle
msg	the message body address

Returns

0 - success
-EINVAL - queue meta data is corrupted

Definition at line 552 of file vmbox.c.

7.1.4.9 vmbox_exit()

```
void vmbox_exit (
            void * handle )
```

Stop working with the virtual message box library.

This function terminates all operations on a given message box instance. The **handle** object which represents this message box instance becomes invalid.

Parameters

handle	the message box instance handle

Definition at line 368 of file vmbox.c.

7.1.5 Variable Documentation

7.1.5.1 receive

```
vmbox_cb_ops_t::receive
```

call back invoked when receiving messages.

Definition at line 38 of file vmbox.h.

7.1.5.2 on

```
vmbox_cb_ops_t::on
```

call back invoked when link goes online.

Definition at line 36 of file vmbox.h.

7.1.5.3 off

```
vmbox_cb_ops_t::off
```

call back invoked when link goes offline.

Definition at line 37 of file vmbox.h.

7.2 Virtual SMQ API

Data Structures

- struct vsmq_cb_ops_t

    Call back notifications. More...

Typedefs

- typedef uint64_t vsmq_link_id_t

    Virtual link identifier.
- typedef uint32_t vsmq_vm_id_t

    VM identifier.
- typedef void(∗ vsmq_call_back_t) (void ∗cookie)

    This type defines call backs which are invoked when software component using vsmq needs to be notified.

Functions

- int vsmq_lookup_tx (const char ∗name, uint32_t msg_size, uint32_t queue_size, void ∗∗handle)

    Look up a transmission (client) virtual link endpoint with a given name.
- int vsmq_lookup_rx (const char ∗name, uint32_t msg_size, uint32_t queue_size, void ∗∗handle)

    Look up a reception (server) virtual link endpoint with a given name.
- int vsmq_lookup_next (void ∗handle, void ∗∗next_handle)

    Lookup next virtual link endpoint of the same name and direction.
- void vsmq_cb_register (void ∗handle, vsmq_cb_ops_t ∗ops, void ∗cookie)

    Register call back operations.
- int vsmq_start (void ∗handle)

    Start communication on a given virtual link.
- bool vsmq_link_is_online (void ∗handle)

    Check whether the virtual link state is online.
- vsmq_link_id_t vsmq_link_id (void ∗handle)

    Get a virtual link ID.
- vsmq_vm_id_t vsmq_my_vmid (void ∗handle)

    Get ID of VM on which I am running.
- vsmq_vm_id_t vsmq_peer_vmid (void ∗handle)

    Get peer VM ID.
- int vsmq_msg_allocate (void ∗handle, void ∗∗msg)

    Allocate a transmission message.
- int vsmq_msg_allocate_early (void ∗handle, void ∗∗msg)

    Allocate a transmission message for early sending.
- int vsmq_msg_send (void ∗handle, void ∗msg)

    Send message.
- int vsmq_msg_send_early (void ∗handle, void ∗msg)

    Send message early.
- int vsmq_msg_receive (void ∗handle, void ∗∗msg)

    Receive message.
- int vsmq_msg_free (void ∗handle, void ∗msg)

    Free received message.
- void vsmq_stop (void ∗handle)

    Stop communication on a given virtual link.
- void vsmq_free (void ∗handle)

    Release a given virtual link.

Variables

- vsmq_call_back_t vsmq_cb_ops_t::sysconf

    This notification call back is invoked on each peer (remote) endpoint state transition. Typically, the vsmq_link_is_online function is used in the call back logic in order to test the current virtual link state.
- vsmq_call_back_t vsmq_cb_ops_t::receive

    This notification call back is invoked each time a new message is received. Typically, the vsmq_msg_receive function is used in the call back logic in order to dequeue incoming messages.

7.2.1 Description

7.2.2 Data Structure Documentation

7.2.2.1 struct vsmq_cb_ops_t

Call back notifications.

Definition at line 33 of file vsmq.h.

Data Fields

vsmq_call_back_t	sysconf	This notification call back is invoked on each peer (remote) endpoint state transition. Typically, the vsmq_link_is_online function is used in the call back logic in order to test the current virtual link state.
vsmq_call_back_t	receive	This notification call back is invoked each time a new message is received. Typically, the vsmq_msg_receive function is used in the call back logic in order to dequeue incoming messages.

7.2.3 Typedef Documentation

7.2.3.1 vsmq_call_back_t

```
typedef void(* vsmq_call_back_t)(void *cookie)
```

This type defines call backs which are invoked when software component using vsmq needs to be notified.

Parameters

cookie	caller cookie given at call back registration time

Definition at line 31 of file vsmq.h.

7.2.4 Function Documentation

7.2.4.1 vsmq_lookup_tx()

```
int vsmq_lookup_tx (
            const char * name,
            uint32_t msg_size,
            uint32_t queue_size,
            void ** handle )
```

Look up a transmission (client) virtual link endpoint with a given name.

The caller invokes this function first in order to begin the enumeration of existing virtual links. This function searches for a first transmission (client) virtual link endpoint matching a given **name** string. On success, the function returns a zero value and stores a queue handle at address provided by the **handle** argument. A zero handle means that the virtual link with a given name doesn't exist. A non-zero handle designates a transmission queue instance and it should be passed back in all subsequent invocations.

The **msg_size** and **queue_size** arguments should have the same values on both virtual link endpoints.

Parameters

name	the **compatible** property string of the virtual link node
msg_size	the message size
queue_size	the transmission queue size (must be a power of 2)
handle	the address where a queue handle is returned on success

Returns

0 - success
-EINVAL - **queue_size** is not a power of 2
-ENOMEM - no enough memory to create a queue descriptor

Definition at line 753 of file vsmq.c.

7.2.4.2 vsmq_lookup_rx()

```
int vsmq_lookup_rx (
            const char * name,
            uint32_t msg_size,
            uint32_t queue_size,
            void ** handle )
```

Look up a reception (server) virtual link endpoint with a given name.

The caller invokes this function first in order to begin the enumeration of existing virtual links. This function searches for a first reception (server) virtual link endpoint matching a given **name** string. On success, the function returns a zero value and stores a queue handle at address provided by the **handle** argument. A zero handle means that the virtual link with a given name doesn't exist. A non-zero handle designates a reception queue instance and it should be passed back in all subsequent invocations.

The **msg_size** and **queue_size** arguments should have the same values on both virtual link endpoints.

Parameters

name	the **compatible** property string in the virtual link node
msg_size	the message size
queue_size	the reception queue size (must be a power of 2)
handle	the address where a queue handle is returned on success

Returns

0 - success
-EINVAL - **queue_size** is not a power of 2
-ENOMEM - no enough memory to create a queue descriptor

Definition at line 760 of file vsmq.c.

7.2.4.3 vsmq_lookup_next()

```
int vsmq_lookup_next (
            void * handle,
            void ** next_handle )
```

Lookup next virtual link endpoint of the same name and direction.

The caller invokes this function in order to continue enumeration of the existing virtual links. This function is searching for a next virtual link endpoint matching the name and direction specified by the **handle** argument. The caller should use the queue handle which has been obtained on the previous enumeration step. On success, the function returns a zero value and stores a queue handle at address provided by the **next_handle** argument. A zero handle means the end of enumeration process, in other words, there is no more virtual link with corresponding name and direction. A non-zero handle designates a queue instance and it should be passed back in all subsequent invocations.

Parameters

handle	the current queue instance handle
next_handle	the address where the next queue handle is returned on success

Returns

0 - success
-ENOMEM - no enough memory to create a queue descriptor

Definition at line 767 of file vsmq.c.

7.2.4.4 vsmq_cb_register()

```
void vsmq_cb_register (
            void * handle,
```

```
            vsmq_cb_ops_t * ops,
            void * cookie )
```

Register call back operations.

This function registers the caller notification call backs. The **cookie** argument is an opaque value for the library and it will just be passed back to the caller as a call back argument. This makes it possible for the caller to associate its own descriptor to the queue instance and then to get this descriptor available in the notification call back.

In order to don't miss a call back notification, it is recommended to connect the call back operations prior to the vsmq_start invocation.

Parameters

handle	the queue instance handle
ops	the structure defining the notification call backs
cookie	the caller descriptor associated with the queue instance

Definition at line 841 of file vsmq.c.

7.2.4.5 vsmq_start()

```
int vsmq_start (
            void * handle )
```

Start communication on a given virtual link.

This function allocates all persistent resources needed for the communication:

- persistent shared memory (PMEM)
- persistent interrupt

and, in the case of success, puts a given message queue into a working state. This function also performs a handshake with the peer endpoint in order to signal about the virtual link state transition.

Typically, vsmq_start function invocation on one endpoint causes vsmq_cb_ops_t::sysconf call back invocation on another endpoint. In such a way, the remote side is notified about readiness for the communication.

Parameters

handle	the queue instance handle

Returns

0 - success
-ENOMEM - no enough persistent or local memory resources

Definition at line 776 of file vsmq.c.

7.2.4.6 vsmq_link_is_online()

```
bool vsmq_link_is_online (
            void * handle )
```

Check whether the virtual link state is online.

This function is typically used in the vsmq_cb_ops_t::sysconf call back logic in order to test the current state of virtual link and perform appropriate actions accordingly. The virtual link is online when both endpoints are ready for the communication.

Parameters

handle	the queue instance handle

Returns

true - virtual link is online
false - virtual link is offline

Definition at line 850 of file vsmq.c.

7.2.4.7 vsmq_link_id()

```
vsmq_link_id_t vsmq_link_id (
            void * handle )
```

Get a virtual link ID.

This function returns a local virtual link identifier. Note that this identifier is not indented to be globally unique and therefore it is provided for the local usage only. It is particularly useful in order to identify endpoints of a same loop back virtual link. Such a loop back bi-directional VSMQ connection is represented by two identical loop back virtual links (one in each direction) and only the virtual link identifier allows to distinguish them.

Parameters

handle	the queue instance handle

Returns

64-bit local virtual link identifier

Definition at line 859 of file vsmq.c.

7.2.4.8 vsmq_my_vmid()

```
vsmq_vm_id_t vsmq_my_vmid (
            void * handle )
```

Get ID of VM on which I am running.

This function returns identifier of the virtual machine on which this code is executed.

Parameters

handle	the queue instance handle

Returns

my (local) virtual machine identifier

Definition at line 867 of file vsmq.c.

7.2.4.9 vsmq_peer_vmid()

```
vsmq_vm_id_t vsmq_peer_vmid (
            void * handle )
```

Get peer VM ID.

This function returns identifier of the virtual machine to which another endpoint of the virtual link is connected.

Parameters

handle	the queue instance handle

Returns

peer (remote) virtual machine identifier

Definition at line 875 of file vsmq.c.

7.2.4.10 vsmq_msg_allocate()

```
int vsmq_msg_allocate (
            void * handle,
            void ** msg )
```

Allocate a transmission message.

This function allocates a transmission message for a given queue instance. A pointer to the message body is stored at the address specified by the **msg** argument. Once a message is successfully allocated and filled in with payload data, the caller should enqueue the message by invoking vsmq_msg_send function.

Note that currently there is no way to drop an allocated message without sending it.

Parameters

handle	the queue instance handle
msg	the address where a pointer to the allocated message body is stored

Returns

0 - success
-ENOTCONN - virtual link is offline
-EAGAIN - no more free messages (the transmission queue is full)
-EINVAL - queue meta data is corrupted

Definition at line 883 of file vsmq.c.

7.2.4.11 vsmq_msg_allocate_early()

```
int vsmq_msg_allocate_early (
            void * handle,
            void ** msg )
```

Allocate a transmission message for early sending.

This function is similar to the vsmq_msg_allocate one. The only difference is that the library allows to allocate a message when the virtual link is still offline but the local endpoint is already ready. It should be understood that the virtual link is online when both endpoints are ready for the communication. Thus, this function allows for an early message allocation while the remote endpoint is not ready yet.

Parameters

handle	the queue instance handle
msg	the address where a pointer to the allocated message body is stored

Returns

0 - success
-ENOTCONN - local virtual link endpoint is not ready
-EAGAIN - no more free messages (the transmission queue is full)
-EINVAL - queue meta data is corrupted

Definition at line 898 of file vsmq.c.

7.2.4.12 vsmq_msg_send()

```
int vsmq_msg_send (
            void * handle,
            void * msg )
```

Send message.

This function puts a given message to a given transmission queue. The **msg** argument points to the message body previously allocated by either vsmq_msg_allocate or vsmq_msg_allocate_early.

Parameters

handle	the queue instance handle
msg	the message address

Returns

0 - success
-ENOTCONN - virtual link is offline
-EINVAL - queue meta data is corrupted

Definition at line 913 of file vsmq.c.

7.2.4.13 vsmq_msg_send_early()

```
int vsmq_msg_send_early (
            void * handle,
            void * msg )
```

Send message early.

This function is similar to the vsmq_msg_send one. The only difference is that the library allows to enqueue a message when the virtual link is still offline but the local endpoint is already ready. It should be understood that the virtual link is online only when both endpoints are ready for the communication. Thus, this function allows for an early message sending while the remote endpoint is not ready yet. This message will be received once the remote endpoint becomes ready.

Parameters

handle	the queue instance handle
msg	the message address

Returns

0 - success
-ENOTCONN - local virtual link endpoint is not ready
-EINVAL - queue meta data is corrupted

Definition at line 928 of file vsmq.c.

7.2.4.14 vsmq_msg_receive()

```
int vsmq_msg_receive (
            void * handle,
            void ** msg )
```

Receive message.

This function is typically invoked by the vsmq_cb_ops_t::receive call back logic in order to dequeue incoming messages. A pointer to the message body is stored at the address specified by the **msg** argument. Once the message is processed the caller should free the message by invoking vsmq_msg_free function.

Parameters

handle	the queue instance handle
msg	the address where a pointer to the received message body is stored

Returns

0 - success
-ENOMSG - no more messages (the reception queue is empty)
-EINVAL - queue meta data is corrupted

Definition at line 943 of file vsmq.c.

7.2.4.15 vsmq_msg_free()

```
int vsmq_msg_free (
            void * handle,
            void * msg )
```

Free received message.

This function release a given message. The **msg** argument points to the message body previously obtained by vsmq_msg_receive.

Parameters

handle	the queue instance handle
msg	the message address

Returns

0 - success
-EINVAL - queue meta data is corrupted

Definition at line 955 of file vsmq.c.

7.2.4.16 vsmq_stop()

```
void vsmq_stop (
            void * handle )
```

Stop communication on a given virtual link.

This function stops communication on a given queue instance and sets the local endpoint to unready state. Typically, vsmq_stop function invocation on one endpoint causes vsmq_cb_ops_t::sysconf call back invocation on another endpoint. In such a way, the remote side is notified about the communication termination.

Parameters

handle	the queue instance handle

Definition at line 967 of file vsmq.c.

7.2.4.17 vsmq_free()

```
void vsmq_free (
            void * handle )
```

Release a given virtual link.

This function terminates all operations on a given queue instance. The **handle** object which represents this queue instance becomes invalid. The queue should be stopped by vsmq_stop prior to releasing.

Parameters

handle	the queue instance handle

Definition at line 984 of file vsmq.c.

7.2.5 Variable Documentation

7.2.5.1 sysconf

```
vsmq_cb_ops_t::sysconf
```

This notification call back is invoked on each peer (remote) endpoint state transition. Typically, the vsmq_link_is_online function is used in the call back logic in order to test the current virtual link state.

Definition at line 34 of file vsmq.h.

7.2.5.2 receive

```
vsmq_cb_ops_t::receive
```

This notification call back is invoked each time a new message is received. Typically, the vsmq_msg_receive function is used in the call back logic in order to dequeue incoming messages.

Definition at line 35 of file vsmq.h.

7.3 Virtual dmaheap API

Chapters

- vDMAHEAP Userspace API Introduction
- Global buffer identifier
- Credentials
- Version numbers
- Capabilities
- Commands

Files

- file vdmaheap_stats_uapi.h

  virtual dmaheap (vdmaheap) driver - userspace stats API
- file vdmaheap_uapi.h

  virtual DMAHEAP (vDMAHEAP) driver - userspace API

Data Structures

- struct vdmaheap_stats_dev

  Statistics command's parameters. More...
- struct vdmaheap_stats_vrpc
- union vdmaheap_stats_res
- struct vdmaheap_stats_arg
- union vdmaheap_stats_data

Macros

- #define VDMAHEAP_IOC_STATS _IOWR(VDMAHEAP_IOC_MAGIC, 55, union vdmaheap_stats_data)

  Driver statistics command.
- #define VDMAHEAP_IOC_MAGIC DMA_HEAP_IOC_MAGIC

  driver commands's magic number

Enumerations

- enum vdmaheap_stats_type {
  VDMAHEAP_STATS_DEV,
  VDMAHEAP_STATS_VRPC_BE,
  VDMAHEAP_STATS_VRPC_FE }

7.3.1 Description

7.3.2 Data Structure Documentation

7.3.2.1 struct vdmaheap_stats_dev

Statistics command's parameters.

This data structure is used as the argument of the VDMAHEAP_IOC_STATS command.

Definition at line 191 of file vdmaheap_stats_uapi.h.

Data Fields

__u64	client_create	[out] number of clients created
__u64	client_destroy	[out] number of clients destroyed
__u64	export_create	[out] number of exports objects created
__u64	export_destroy	[out] number of exports objects destroyed. When in the idle state, export_create and export_destroy counter values are identical
__u64	import_create	[out] number of imports objects created
__u64	import_destroy	[out] number of imports objects destroyed. When in the idle state, import_create and import_destroy counter values are identical
__u64	dmabuf_create	[out] number of dma_buf objects created
__u64	dmabuf_destroy	[out] number of dma_buf objects destroyed. When in the idle state, dmabuf_create and dmabuf_destroy counter values are identical

7.3.2.2 struct vdmaheap_stats_vrpc

Definition at line 201 of file vdmaheap_stats_uapi.h.

7.3.2.3 union vdmaheap_stats_res

Definition at line 230 of file vdmaheap_stats_uapi.h.

7.3.2.4 struct vdmaheap_stats_arg

Definition at line 239 of file vdmaheap_stats_uapi.h.

Data Fields

__u32	type	[in] the type of stats to retrieve in enum vdmaheap_stats_type range
__u32	peer	[in] peer domain id of the vlink whose stats are retrieved

7.3.2.5 union vdmaheap_stats_data

Definition at line 243 of file vdmaheap_stats_uapi.h.

7.3.3 Macro Definition Documentation

7.3.3.1 VDMAHEAP_IOC_STATS

```
#define VDMAHEAP_IOC_STATS _IOWR(VDMAHEAP_IOC_MAGIC, 55, union vdmaheap_stats_data)
```

Driver statistics command.

This command returns statistics counters maintained by the vdmaheap driver. This enables a vdmaheap client to determine how many objects have been created and destroyed by the vdmaheap driver since its initialization.

Definition at line 184 of file vdmaheap_stats_uapi.h.

7.3.3.2 VDMAHEAP_IOC_MAGIC

```
#define VDMAHEAP_IOC_MAGIC DMA_HEAP_IOC_MAGIC
```

driver commands's magic number

This statement provides the "magic number" of vDMAHEAP's ioctl commands. Commands that do not match this number are rejected by the vDMAHEAP driver. Note that vDMAHEAP re-uses the same number as the native DMAHEAP driver.

Definition at line 131 of file vdmaheap_uapi.h.

7.3.4 Enumeration Type Documentation

7.3.4.1 vdmaheap_stats_type

```
enum vdmaheap_stats_type
```

Enumerator

VDMAHEAP_STATS_DEV	retrieve device related stats
VDMAHEAP_STATS_VRPC_BE	retrieve vRPC back-end (exporter) related stats
VDMAHEAP_STATS_VRPC_FE	retrieve vRPC front-end (importer) related stats

Definition at line 234 of file vdmaheap_stats_uapi.h.

7.4 vDMAHEAP Userspace API Introduction

7.4.1 Introduction

vDMAHEAP enhances the functionalities offered by the DMAHEAP driver, a generalized memory manager originally introduced by Google as part of Android.

Specifically, the vDMAHEAP driver provides in-kernel support for the secure sharing of DMAHEAP buffers between userspace processes. In the context of virtualized systems, vDMAHEAP extends this concept to embrace the sharing of DMAHEAP buffers between guest operating systems that execute in separate virtual machines, or domains.

The vdmaheap_uapi.h header defines the programming interface exposed by the vDMAHEAP driver to userspace clients.

7.4.2 Background

The DMAHEAP memory allocator enables userspace clients to allocate buffers from memory regions presented as heaps. Some of these heaps are pre-reserved at boot time while others merely leverage the kernel's standard memory allocators. Each type of device can be provisioned with a different set of standard and platform-specific DMAHEAP heaps, according to the memory requirements of the device.

Through the API exposed by the /dev/ion device, DMAHEAP is essentially a service for client processes to allocate `DMABUF` buffers that use DMAHEAP memory heaps as their backing storage.

7.4.3 Features

vDMAHEAP implements the sharing of DMAHEAP buffers through a secured userspace API that offers buffer export and buffer import commands. The same API enables to transparently share DMAHEAP buffers between processes that are either hosted by the same guest operating system or by different guests executing in separate domains.

vDMAHEAP does not arbitrate concurrent accesses to the memory exposed by a shared DMAHEAP buffer. Such concurrent acccesses are expected to be policed by some external token management or producer/consumer service. Similarly, vDMAHEAP does not perform by itself any synchronization operation on shared memory. Processes that share a DMAHEAP buffer shall use the msync() system call, or DMABUF's finer-grained DMA_BUF_IOCTL↩
_SYNC command to ensure that they always access the buffer's content in a consistent manner.

vDMAHEAP introduces the notion of global buffer identifiers (GIDs) to identify the DMAHEAP buffers that are exported to other processes. vDMAHEAP's system-wide GIDs uniquely identify DMAHEAP buffers regardless of the domain it was allocated in.

The API exposed by the DMAHEAP driver can be secured by the optional use of buffer credentials. Credentials are opaque data structures that match global buffer identifiers. Buffer credentials consists in an 16-byte UUID set to a random value at the buffer's export time. When vDMAHEAP is set to operate in secure mode, vDMAHEAP clients are required to present these credentials in addition to global identifiers to import DMAHEAP buffers.

7.5 Global buffer identifier

Typedefs

- typedef __u32 vdmaheap_buffer_gid_t
  Global buffer identifier.
- typedef __u32 vion_buffer_gid_t
  Global buffer identifier.

7.5.1 Description

vDMAHEAP's global buffer identifiers (GIDs) are 32- or 64-bit integers, depending on build-time options. They combine a buffer allocation domain identifier with a per-domain local buffer identifier to produce a system-wide, unique buffer identifier.

vION's global buffer identifiers (GIDs) are 32- or 64-bit integers, depending on build-time options. They combine a buffer allocation domain identifier with a per-domain local buffer identifier to produce a system-wide, unique buffer identifier.

7.5.2 Typedef Documentation

7.5.2.1 vdmaheap_buffer_gid_t

```
typedef __u32 vdmaheap_buffer_gid_t
```

Global buffer identifier.

This definition provides the type of the global identifiers (GIDs) that uniquely name the DMAHEAP buffers shared by vDMAHEAP.

Definition at line 121 of file vdmaheap_uapi.h.

7.5.2.2 vion_buffer_gid_t

```
typedef __u32 vion_buffer_gid_t
```

Global buffer identifier.

This definition provides the type of the global identifiers (GIDs) that uniquely name the ION buffers shared by vION.

Definition at line 119 of file vion_uapi.h.

7.6 Credentials

Data Structures

- struct uuid_t

  Buffer credentials. More...

Macros

- #define UUID_SIZE 16

  Length of a UUID's byte array.
- #define UUID_SIZE 16

  Length of a UUID's byte array.

7.6.1 Description

Credentials secure the sharing of the DMAHEAP buffers. They consists in an opaque, 16-byte UUID set to a random value at the buffer's export time. When vDMAHEAP operates in secure mode, vDMAHEAP clients are required to present credentials in addition to global identifiers to import DMAHEAP buffers.

Credentials secure the sharing of the ION buffers. They consists in an opaque, 16-byte UUID set to a random value at the buffer's export time. When vION operates in secure mode, vION clients are required to present credentials in addition to global identifiers to import ION buffers.

7.6.2 Data Structure Documentation

7.6.2.1 struct uuid_t

Buffer credentials.

This definition provides the type of the universally unique identifier (UUIDs) used as credentials to secure the sharing of DMAHEAP buffers.

This definition provides the type of the universally unique identifier (UUIDs) used as credentials to secure the sharing of ION buffers.

Definition at line 156 of file vdmaheap_uapi.h.

Data Fields

__u8	b[UUID_SIZE]	UUID byte array.

7.7 Version numbers

Macros

- #define VDMAHEAP_VERSION_0 0
- #define VDMAHEAP_VERSION_1 1
- #define VDMAHEAP_VERSION_2 2
- #define VDMAHEAP_VERSION_3 3
- #define VDMAHEAP_VERSION_4 4
- #define VDMAHEAP_VERSION VDMAHEAP_VERSION_4

  Userspace API's current version number.
- #define VION_VERSION_0 0
- #define VION_VERSION_1 1
- #define VION_VERSION_2 2
- #define VION_VERSION_3 3
- #define VION_VERSION_4 4
- #define VION_VERSION VION_VERSION_4

  Userspace API's current version number.

7.7.1 Description

These definitions provides version numbers that identify the successive versions of vDMAHEAP's userspace API, as reported by the VDMAHEAP_IOC_VERSION command.

These definitions provides version numbers that identify the successive versions of vION's userspace API, as reported by the VION_IOC_VERSION command.

7.7.2 Macro Definition Documentation

7.7.2.1 VDMAHEAP_VERSION_0

```
#define VDMAHEAP_VERSION_0 0
```

Userspace API version 0

Definition at line 174 of file vdmaheap_uapi.h.

7.7.2.2 VDMAHEAP_VERSION_1

```
#define VDMAHEAP_VERSION_1 1
```

Userspace API version 1

Definition at line 175 of file vdmaheap_uapi.h.

7.7.2.3 VDMAHEAP_VERSION_2

```
#define VDMAHEAP_VERSION_2 2
```

Userspace API version 2

Definition at line 176 of file vdmaheap_uapi.h.

7.7.2.4 VDMAHEAP_VERSION_3

```
#define VDMAHEAP_VERSION_3 3
```

Userspace API version 3

Definition at line 177 of file vdmaheap_uapi.h.

7.7.2.5 VDMAHEAP_VERSION_4

```
#define VDMAHEAP_VERSION_4 4
```

Userspace API version 3

Definition at line 178 of file vdmaheap_uapi.h.

7.7.2.6 VDMAHEAP_VERSION

```
#define VDMAHEAP_VERSION VDMAHEAP_VERSION_4
```

Userspace API's current version number.

This definition provides the version number of vDMAHEAP's current userspace API, as reported by the VDMAHEAP_IOC_VERSION command. This version number is incremented each time the userspace API is modified or enhanced.

Definition at line 187 of file vdmaheap_uapi.h.

7.7.2.7 VION_VERSION_0

```
#define VION_VERSION_0 0
```

Userspace API version 0

Definition at line 172 of file vion_uapi.h.

7.7.2.8 VION_VERSION_1

```
#define VION_VERSION_1 1
```

Userspace API version 1

Definition at line 173 of file vion_uapi.h.

7.7.2.9 VION_VERSION_2

```
#define VION_VERSION_2 2
```

Userspace API version 2

Definition at line 174 of file vion_uapi.h.

7.7.2.10 VION_VERSION_3

```
#define VION_VERSION_3 3
```

Userspace API version 3

Definition at line 175 of file vion_uapi.h.

7.7.2.11 VION_VERSION_4

```
#define VION_VERSION_4 4
```

Userspace API version 3

Definition at line 176 of file vion_uapi.h.

7.7.2.12 VION_VERSION

```
#define VION_VERSION VION_VERSION_4
```

Userspace API's current version number.

This definition provides the version number of vION's current userspace API, as reported by the VION_IOC_VERSION command. This version number is incremented each time the userspace API is modified or enhanced.

Definition at line 185 of file vion_uapi.h.

7.8 Capabilities

Macros

- #define VDMAHEAP_CAP_SECURE (1U << 0)
- #define VDMAHEAP_CAP_LOCAL (1U << 1)
- #define VDMAHEAP_CAP_REMOTE (1U << 2)
- #define VDMAHEAP_CAP_VBB (1U << 3)
- #define VDMAHEAP_CAP_STRICT_IMPORT
- #define VION_CAP_SECURE (1U << 0)
- #define VION_CAP_LOCAL (1U << 1)
- #define VION_CAP_REMOTE (1U << 2)
- #define VION_CAP_VBB (1U << 3)
- #define VION_CAP_STRICT_IMPORT

7.8.1 Description

These definitions describe the capabilities of the vDMAHEAP driver, as reported by the VDMAHEAP_IOC_VERSION command.

These definitions describe the capabilities of the vION driver, as reported by the VION_IOC_VERSION command.

7.8.2 Macro Definition Documentation

7.8.2.1 VDMAHEAP_CAP_SECURE

```
#define VDMAHEAP_CAP_SECURE (1U << 0)
```

The driver operates in secure mode

Definition at line 197 of file vdmaheap_uapi.h.

7.8.2.2 VDMAHEAP_CAP_LOCAL

```
#define VDMAHEAP_CAP_LOCAL (1U << 1)
```

The driver supports domain-local buffer sharing

Definition at line 198 of file vdmaheap_uapi.h.

7.8.2.3 VDMAHEAP_CAP_REMOTE

```
#define VDMAHEAP_CAP_REMOTE (1U << 2)
```

The driver supports inter-domain buffer sharing

Definition at line 200 of file vdmaheap_uapi.h.

7.8.2.4 VDMAHEAP_CAP_VBB

```
#define VDMAHEAP_CAP_VBB (1U << 3)
```

The driver exposes a VBB-compatible API

Definition at line 202 of file vdmaheap_uapi.h.

7.8.2.5 VDMAHEAP_CAP_STRICT_IMPORT

```
#define VDMAHEAP_CAP_STRICT_IMPORT
```

Value:

```
(1U                                                                          \
    << 4)
```

The driver enforces a strict import semantic, where the the validity of gids and credentials is systematically checked, even if the buffer is already locally imported

Definition at line 203 of file vdmaheap_uapi.h.

7.8.2.6 VION_CAP_SECURE

```
#define VION_CAP_SECURE (1U << 0)
```

The driver operates in secure mode

Definition at line 195 of file vion_uapi.h.

7.8.2.7 VION_CAP_LOCAL

```
#define VION_CAP_LOCAL (1U << 1)
```

The driver supports domain-local buffer sharing

Definition at line 196 of file vion_uapi.h.

7.8.2.8 VION_CAP_REMOTE

```
#define VION_CAP_REMOTE (1U << 2)
```

The driver supports inter-domain buffer sharing

Definition at line 198 of file vion_uapi.h.

7.8.2.9 VION_CAP_VBB

```
#define VION_CAP_VBB (1U << 3)
```

The driver exposes a VBB-compatible API

Definition at line 200 of file vion_uapi.h.

7.8.2.10 VION_CAP_STRICT_IMPORT

```
#define VION_CAP_STRICT_IMPORT
```

Value:

```
(1U                                                                                           \
    << 4)
```

The driver enforces a strict import semantic, where the the validity of gids and credentials is systematically checked, even if the buffer is already locally imported

Definition at line 201 of file vion_uapi.h.

7.9 Commands

Data Structures

- struct vdmaheap_version_data

  API version handshake command's argument. More...
- struct vdmaheap_export_data

  Export command's argument. More...
- struct vdmaheap_unexport_data

  Unexport command's argument. More...
- struct vdmaheap_import_data

  Import command's argument. More...
- struct vdmaheap_info_data

  Info command's parameters. More...
- struct vdmaheap_link_state

  Info command's parameters. More...
- struct vion_version_data

  API version handshake command's argument. More...
- struct vion_export_data

  Export command's argument. More...
- struct vion_unexport_data

  Unexport command's argument. More...
- struct vion_import_data

  Import command's argument. More...
- struct vion_info_data

  Info command's parameters. More...
- struct vion_link_state

  Info command's parameters. More...

Macros

- #define VDMAHEAP_IOC_VERSION _IOWR(VDMAHEAP_IOC_MAGIC, 54, struct vdmaheap_version_data)

  API version handshake command.
- #define VDMAHEAP_IOC_EXPORT _IOWR(VDMAHEAP_IOC_MAGIC, 50, struct vdmaheap_export_data)

  Buffer export command.
- #define VDMAHEAP_IOC_UNEXPORT _IOWR(VDMAHEAP_IOC_MAGIC, 51, struct vdmaheap_unexport_data)

  Buffer unexport command.
- #define VDMAHEAP_IOC_IMPORT _IOWR(VDMAHEAP_IOC_MAGIC, 52, struct vdmaheap_import_data)

  Buffer import command.
- #define VDMAHEAP_IOC_INFO _IOWR(VDMAHEAP_IOC_MAGIC, 53, struct vdmaheap_info_data)

  Buffer information command.
- #define VDMAHEAP_IOC_LINK_STATE _IOWR(VDMAHEAP_IOC_MAGIC, 55, struct vdmaheap_info_data)

  Read/wait for vDMAHEAP connection with its peer backends.
- #define VION_IOC_VERSION _IOWR(VION_IOC_MAGIC, 54, struct vion_version_data)

  API version handshake command.
- #define VION_IOC_EXPORT _IOWR(VION_IOC_MAGIC, 50, struct vion_export_data)

  Buffer export command.
- #define VION_IOC_UNEXPORT _IOWR(VION_IOC_MAGIC, 51, struct vion_unexport_data)

  Buffer unexport command.
- #define VION_IOC_IMPORT _IOWR(VION_IOC_MAGIC, 52, struct vion_import_data)

Buffer import command.
- #define VION_IOC_INFO _IOWR(VION_IOC_MAGIC, 53, struct vion_info_data)

  Buffer information command.
- #define VION_IOC_LINK_STATE _IOWR(VION_IOC_MAGIC, 55, struct vion_info_data)

  Read/wait for vION connection with its peer backends.

Functions

- static int vdmaheap_gid_origin (vdmaheap_buffer_gid_t gid)

  Buffer origin domain.
- static int vion_gid_origin (vion_buffer_gid_t gid)

  Buffer origin domain.

7.9.1 Description

These definitions describe the commands offered by the vDMAHEAP driver.

These definitions describe the commands offered by the vION driver.

7.9.2 Data Structure Documentation

7.9.2.1 struct vdmaheap_version_data

API version handshake command's argument.

This data structure is used as the argument of the VDMAHEAP_IOC_VERSION command.

Definition at line 231 of file vdmaheap_uapi.h.

Data Fields

__u32	version	[in] API version expected by the client [out] API version supported by the driver
__u32	caps	[out] Capabilities of the driver
__u64	reserved	Reserved for future use

7.9.2.2 struct vdmaheap_export_data

Export command's argument.

This data structure is used as the argument of the VDMAHEAP_IOC_EXPORT command.

Definition at line 255 of file vdmaheap_uapi.h.

Data Fields

__u32	fd	[in] DMAHEAP buffer fd to export

Data Fields

vdmaheap_buffer_gid_t	gid	[out] global buffer identifier
uuid_t	cred	[out] buffer credentials
__u32	size	[out] buffer size

7.9.2.3 struct vdmaheap_unexport_data

Unexport command's argument.

This data structure is used as the argument of the VDMAHEAP_IOC_UNEXPORT command.

Definition at line 278 of file vdmaheap_uapi.h.

Data Fields

vdmaheap_buffer_gid_t	gid	[in] global buffer identifier
uuid_t	cred	[in] buffer credentials

7.9.2.4 struct vdmaheap_import_data

Import command's argument.

This data structure is used as the argument of the VDMAHEAP_IOC_IMPORT command.

Definition at line 298 of file vdmaheap_uapi.h.

Data Fields

__u32	fd	[out] imported DMAHEAP buffer's file descriptor
vdmaheap_buffer_gid_t	gid	[in] global buffer identifier
uuid_t	cred	[in] buffer credentials
__u32	size	[out] buffer size

7.9.2.5 struct vdmaheap_info_data

Info command's parameters.

This data structure is used as the argument of the VDMAHEAP_IOC_INFO command.

Definition at line 336 of file vdmaheap_uapi.h.

Data Fields

vdmaheap_buffer_gid_t	gid	[in] global buffer identifier
uuid_t	cred	[in] buffer credentials
__u64	flags	[out] buffer flags
__u32	size	[out] buffer size
const char *	heap_name	[out] dma heap identifier

7.9.2.6 struct vdmaheap_link_state

Info command's parameters.

This data structure is used as the argument of the VDMAHEAP_IOC_LINK_STATE command.

Definition at line 360 of file vdmaheap_uapi.h.

Data Fields

__u32	wait	[in] boolean value: 0: read link state, 1: wait for link state to equal value
__u32	mask	[in] mask of domains we are interrested in the state
__u32	value	[in/out] if wait=0, the read connection state, if wait=1, the expected connection state

7.9.2.7 struct vion_version_data

API version handshake command's argument.

This data structure is used as the argument of the VION_IOC_VERSION command.

Definition at line 229 of file vion_uapi.h.

Data Fields

__u32	version	[in] API version expected by the client [out] API version supported by the driver
__u32	caps	[out] Capabilities of the driver
__u64	reserved	Reserved for future use

7.9.2.8 struct vion_export_data

Export command's argument.

This data structure is used as the argument of the VION_IOC_EXPORT command.

Definition at line 253 of file vion_uapi.h.

Data Fields

__u32	fd	[in] ION buffer fd to export
vion_buffer_gid_t	gid	[out] global buffer identifier
uuid_t	cred	[out] buffer credentials
__u32	size	[out] buffer size

7.9.2.9 struct vion_unexport_data

Unexport command's argument.

This data structure is used as the argument of the VION_IOC_UNEXPORT command.

Definition at line 276 of file vion_uapi.h.

Data Fields

vion_buffer_gid_t	gid	[in] global buffer identifier
uuid_t	cred	[in] buffer credentials

7.9.2.10 struct vion_import_data

Import command's argument.

This data structure is used as the argument of the VION_IOC_IMPORT command.

Definition at line 296 of file vion_uapi.h.

Data Fields

__u32	fd	[out] imported ION buffer's file descriptor
vion_buffer_gid_t	gid	[in] global buffer identifier
uuid_t	cred	[in] buffer credentials
__u32	size	[out] buffer size

7.9.2.11 struct vion_info_data

Info command's parameters.

This data structure is used as the argument of the VION_IOC_INFO command.

Definition at line 334 of file vion_uapi.h.

Data Fields

vion_buffer_gid_t	gid	[in] global buffer identifier
uuid_t	cred	[in] buffer credentials
__u64	flags	[out] buffer flags
__u32	size	[out] buffer size
__u32	heap_id	[out] ION heap identifier

7.9.2.12 struct vion_link_state

Info command's parameters.

This data structure is used as the argument of the VION_IOC_LINK_STATE command.

Definition at line 358 of file vion_uapi.h.

Data Fields

__u32	wait	[in] boolean value: 0: read link state, 1: wait for link state to equal value
__u32	mask	[in] mask of domains we are interrested in the state
__u32	value	[in/out] if wait=0, the read connection state, if wait=1, the expected connection state

7.9.3 Macro Definition Documentation

7.9.3.1 VDMAHEAP_IOC_VERSION

```
#define VDMAHEAP_IOC_VERSION _IOWR(VDMAHEAP_IOC_MAGIC, 54, struct vdmaheap_version_data)
```

API version handshake command.

This command enables the vDMAHEAP client and driver to exchange the version number of the vDMAHEAP interface they respectively support. This enables both the client and the driver to determine if they can interoperate. Also returned are flags describing the capabilities of the driver.

Definition at line 224 of file vdmaheap_uapi.h.

7.9.3.2 VDMAHEAP_IOC_EXPORT

```
#define VDMAHEAP_IOC_EXPORT _IOWR(VDMAHEAP_IOC_MAGIC, 50, struct vdmaheap_export_data)
```

Buffer export command.

This command enables a vDMAHEAP client to export a ION buffer. The export operation provides the client with the global identifier and the credentials of the exported DMAHEAP buffer. This global identifier and credentials can be passed to another process for the purpose of importing (i.e. sharing) the DMAHEAP buffer.

Definition at line 247 of file vdmaheap_uapi.h.

7.9.3.3 VDMAHEAP_IOC_UNEXPORT

```
#define VDMAHEAP_IOC_UNEXPORT _IOWR(VDMAHEAP_IOC_MAGIC, 51, struct vdmaheap_unexport_data)
```

Buffer unexport command.

This command enables the vDMAHEAP client to unimport a DMAHEAP buffer, i.e. to revoke a previous export operation. Unexporting a DMAHEAP buffer prevents new clients from importing it. In contrast, the DMAHEAP buffer remain shared with processes that have already imported it.

Definition at line 271 of file vdmaheap_uapi.h.

7.9.3.4 VDMAHEAP_IOC_IMPORT

```
#define VDMAHEAP_IOC_IMPORT _IOWR(VDMAHEAP_IOC_MAGIC, 52, struct vdmaheap_import_data)
```

Buffer import command.

This command enables the vDMAHEAP client to import a DMAHEAP buffer. The import operation provides the client with a file descriptor pointing at the imported DMAHEAP buffer. This file descriptor can be used to map the buffer in userspace and to perform memory synchronization operations on the buffer.

Definition at line 291 of file vdmaheap_uapi.h.

7.9.3.5 VDMAHEAP_IOC_INFO

```
#define VDMAHEAP_IOC_INFO _IOWR(VDMAHEAP_IOC_MAGIC, 53, struct vdmaheap_info_data)
```

Buffer information command.

This command returns information regarding an exported DMAHEAP buffer. This enables a vDMAHEAP client to get the details of a DMAHEAP buffer imported from another userspace process.

Definition at line 329 of file vdmaheap_uapi.h.

7.9.3.6 VDMAHEAP_IOC_LINK_STATE

```
#define VDMAHEAP_IOC_LINK_STATE _IOWR(VDMAHEAP_IOC_MAGIC, 55, struct vdmaheap_info_data)
```

Read/wait for vDMAHEAP connection with its peer backends.

When wait is false, this command reads the connection state with the peer backends. When wait is true, it waits for the connection state to reach a given value. The connection state with peer backends is a 32 bit bitmap with a bit per domain. A bit at 1 means the connection is opened, a bit at 0 means the connection is closed.

Definition at line 353 of file vdmaheap_uapi.h.

7.9.3.7 VION_IOC_VERSION

```
#define VION_IOC_VERSION _IOWR(VION_IOC_MAGIC, 54, struct vion_version_data)
```

API version handshake command.

This command enables the vION client and driver to exchange the version number of the vION interface they respectively support. This enables both the client and the driver to determine if they can interoperate. Also returned are flags describing the capabilities of the driver.

Definition at line 222 of file vion_uapi.h.

7.9.3.8 VION_IOC_EXPORT

```
#define VION_IOC_EXPORT _IOWR(VION_IOC_MAGIC, 50, struct vion_export_data)
```

Buffer export command.

This command enables a vION client to export a ION buffer. The export operation provides the client with the global identifier and the credentials of the exported ION buffer. This global identifier and credentials can be passed to another process for the purpose of importing (i.e. sharing) the ION buffer.

Definition at line 245 of file vion_uapi.h.

7.9.3.9 VION_IOC_UNEXPORT

```
#define VION_IOC_UNEXPORT _IOWR(VION_IOC_MAGIC, 51, struct vion_unexport_data)
```

Buffer unexport command.

This command enables the vION client to unimport a ION buffer, i.e. to revoke a previous export operation. Unexporting a ION buffer prevents new clients from importing it. In contrast, the ION buffer remain shared with processes that have already imported it.

Definition at line 269 of file vion_uapi.h.

7.9.3.10 VION_IOC_IMPORT

```
#define VION_IOC_IMPORT _IOWR(VION_IOC_MAGIC, 52, struct vion_import_data)
```

Buffer import command.

This command enables the vION client to import a ION buffer. The import operation provides the client with a file descriptor pointing at the imported ION buffer. This file descriptor can be used to map the buffer in userspace and to perform memory synchronization operations on the buffer.

Definition at line 289 of file vion_uapi.h.

7.9.3.11 VION_IOC_INFO

```
#define VION_IOC_INFO _IOWR(VION_IOC_MAGIC, 53, struct vion_info_data)
```

Buffer information command.

This command returns information regarding an exported ION buffer. This enables a vION client to get the details of a ION buffer imported from another userspace process.

Definition at line 327 of file vion_uapi.h.

7.9.3.12 VION_IOC_LINK_STATE

```
#define VION_IOC_LINK_STATE _IOWR(VION_IOC_MAGIC, 55, struct vion_info_data)
```

Read/wait for vION connection with its peer backends.

When wait is false, this command reads the connection state with the peer backends. When wait is true, it waits for the connection state to reach a given value. The connection state with peer backends is a 32 bit bitmap with a bit per domain. A bit at 1 means the connection is opened, a bit at 0 means the connection is closed.

Definition at line 351 of file vion_uapi.h.

7.9.4 Function Documentation

7.9.4.1 vdmaheap_gid_origin()

```
static int vdmaheap_gid_origin (
            vdmaheap_buffer_gid_t gid ) [inline], [static]
```

Buffer origin domain.

This function returns the identifier of the domain where the buffer was exported.

Parameters

gid	the buffer

Definition at line 312 of file vdmaheap_uapi.h.

7.9.4.2 vion_gid_origin()

```
static int vion_gid_origin (
            vion_buffer_gid_t gid ) [inline], [static]
```

Buffer origin domain.

This function returns the identifier of the domain where the buffer was exported.

Parameters

gid	the buffer

Definition at line 310 of file vion_uapi.h.

7.10 Virtual DMA fence v2 User mode API

Chapters

- vFence2 user space API Introduction
- Global DMA fence identifier
- Commands

Files

- file vfence2_uapi.h
  virtual DMA fence v2 (vFence2) driver - user space API

Macros

- #define VFENCE2_IOC_MAGIC 'F'
  driver command magic number

7.10.1 Description

7.10.2 Macro Definition Documentation

7.10.2.1 VFENCE2_IOC_MAGIC

```
#define VFENCE2_IOC_MAGIC 'F'
```

driver command magic number

This statement provides the "magic number" of vFence2 ioctl commands. Commands that do not match this number are rejected by the vFence2 driver.

Definition at line 112 of file vfence2_uapi.h.

7.11 vFence2 user space API Introduction

7.11.1 Introduction

vFence2 enhances the functionalities offered by the DMA fences with inter-VM distribution. It allows to share a DMA fence between guest operating systems that execute in separate virtual machines, or domains.

The vfence2_uapi.h header defines the ioctl-based programming interface exposed by the vFence2 driver to user space clients. Access to the driver is obtained by opening the /dev/vfence2 special file.

7.11.2 Background

The DMA fence feature allows to give or receive an object to/from another process or driver, while this object is not yet finalized. Whether the object is finalized can be read from the DMA fence, and a recipient can also be notified asynchronously.

7.11.3 Features

vFence2 implements sharing of DMA fences through a user space API that offers DMA fence export and import commands. The API enables to share DMA fences between processes that are hosted by different operating system.

vFence2 introduces the notion of global DMA fence identifiers (GIDs) to identify the DMA fences that are exported to other VMs. These GIDs are VM-relative.

Closing the driver file handle automatically unexports all previously exported DMA fences. Exported DMA fences remain shared with processes that have already imported them.

7.12 Global DMA fence identifier

Macros

- #define VFENCE2_ID_NONE 0

  Non-existing DMA fence identifier.

Typedefs

- typedef __u32 vfence2_id_t

  Exported DMA fence identifier.

7.12.1 Description

vFence2's global DMA fence identifiers (GIDs) are 32-bit integers. They contain a per-domain DMA fence identifier.

7.12.2 Typedef Documentation

7.12.2.1 vfence2_id_t

```
typedef __u32 vfence2_id_t
```

Exported DMA fence identifier.

This definition provides the type of the global identifiers (GIDs) that uniquely name the DMA fences shared by vFence2. vfence2_id_t has the size of an integer, so that it can fit inside an "int fd" field in a structure.

Definition at line 96 of file vfence2_uapi.h.

7.13 Commands

Data Structures

- struct vfence2_export_data

  Export command's argument. More...
- struct vfence2_unexport_data

  Unexport command's argument. More...
- struct vfence2_import_data

  Import command's argument. More...

Macros

- #define VFENCE2_IOC_EXPORT _IOWR(VFENCE2_IOC_MAGIC, 50, struct vfence2_export_data)

  DMA fence export command.
- #define VFENCE2_IOC_UNEXPORT _IOWR(VFENCE2_IOC_MAGIC, 51, struct vfence2_unexport_data)

  DMA fence unexport command.
- #define VFENCE2_IOC_IMPORT _IOWR(VFENCE2_IOC_MAGIC, 52, struct vfence2_import_data)

  DMA fence import command.

7.13.1 Description

These definitions describe the commands offered by the vFence2 driver.

7.13.2 Data Structure Documentation

7.13.2.1 struct vfence2_export_data

Export command's argument.

This data structure is used as the argument of the VFENCE2_IOC_EXPORT command. The target VM must be different from the current VM.

Definition at line 149 of file vfence2_uapi.h.

Data Fields

__s32	fd	[in] DMA fence fd to export
__u32	vmid	[in] VM identifier of target VM
vfence2_id_t	gid	[out] global DMA fence identifier

7.13.2.2 struct vfence2_unexport_data

Unexport command's argument.

This data structure is used as the argument of the VFENCE2_IOC_UNEXPORT command. The gid must have been obtained from a previous VFENCE2_IOC_EXPORT call. The target VM must be different from the current VM.

Definition at line 176 of file vfence2_uapi.h.

Data Fields

__u32	vmid	[in] VM identifier of target VM
vfence2_id_t	gid	[in] global DMA fence identifier

7.13.2.3 struct vfence2_import_data

Import command's argument.

This data structure is used as the argument of the VFENCE2_IOC_IMPORT command. A gid returned by VFENCE2_IOC_EXPORT can only be imported once. The source VM must be different from the current VM.

Definition at line 201 of file vfence2_uapi.h.

Data Fields

__s32	fd	[out] imported DMA fence's file descriptor
__u32	vmid	[in] VM identifier of source VM
vfence2_id_t	gid	[in] global DMA fence identifier

7.13.3 Macro Definition Documentation

7.13.3.1 VFENCE2_IOC_EXPORT

```
#define VFENCE2_IOC_EXPORT _IOWR(VFENCE2_IOC_MAGIC, 50, struct vfence2_export_data)
```

DMA fence export command.

This command enables a vFence2 client to export a DMA fence towards a specific VM. The export operation provides the client with the global identifier of the exported DMA fence. This global identifier can be passed to another process in another VM for the purpose of importing (i.e. sharing) the DMA fence. If the specified VM is not yet completely initialized, and the driver file is set to non-blocking mode, error EAGAIN will be returned. If the file is set to blocking mode, the driver will wait until the DMA fence is signaled before returning from this call. In this case, the call will succeed but a zero gid (VFENCE2_ID_NONE) will be returned.

Definition at line 138 of file vfence2_uapi.h.

7.13.3.2 VFENCE2_IOC_UNEXPORT

```
#define VFENCE2_IOC_UNEXPORT _IOWR(VFENCE2_IOC_MAGIC, 51, struct vfence2_unexport_data)
```

DMA fence unexport command.

This command enables the vFence2 client to unexport a DMA fence, i.e. to revoke a previous export operation. It is only supposed to be used in error processing paths. Unexporting a DMA fence prevents new clients from importing it. In contrast, the DMA fence remains shared with processes that have already imported it. However, it may be signaled.

Definition at line 165 of file vfence2_uapi.h.

7.13.3.3 VFENCE2_IOC_IMPORT

```
#define VFENCE2_IOC_IMPORT _IOWR(VFENCE2_IOC_MAGIC, 52, struct vfence2_import_data)
```

DMA fence import command.

This command enables the vFence2 client to import a DMA fence given a valid (non-zero) gid. The import operation provides the client with a file descriptor pointing at the imported DMA fence. This file descriptor can be passed to drivers.

Definition at line 190 of file vfence2_uapi.h.

7.14 Virtual ION API

Chapters

- vION Userspace API Introduction
- Global buffer identifier
- Credentials
- Version numbers
- Capabilities
- Commands

Files

- file vion_stats_uapi.h

  virtual ION (vION) driver - userspace stats API
- file vion_uapi.h

  virtual ION (vION) driver - userspace API

Data Structures

- struct vion_stats_dev

  Statistics command's parameters. More...
- struct vion_stats_vrpc
- union vion_stats_res
- struct vion_stats_arg
- union vion_stats_data

Macros

- #define VION_IOC_STATS _IOWR(VION_IOC_MAGIC, 55, union vion_stats_data)

  Driver statistics command.
- #define VION_IOC_MAGIC ION_IOC_MAGIC

  driver commands's magic number

Enumerations

- enum vion_stats_type {
  VION_STATS_DEV,
  VION_STATS_VRPC_BE,
  VION_STATS_VRPC_FE }

7.14.1 Description

7.14.2 Data Structure Documentation

7.14.2.1 struct vion_stats_dev

Statistics command's parameters.

This data structure is used as the argument of the VION_IOC_STATS command.

Definition at line 191 of file vion_stats_uapi.h.

Data Fields

__u64	client_create	[out] number of clients created
__u64	client_destroy	[out] number of clients destroyed
__u64	export_create	[out] number of exports objects created
__u64	export_destroy	[out] number of exports objects destroyed. When in the idle state, export_create and export_destroy counter values are identical
__u64	import_create	[out] number of imports objects created
__u64	import_destroy	[out] number of imports objects destroyed. When in the idle state, import_create and import_destroy counter values are identical
__u64	dmabuf_create	[out] number of dma_buf objects created
__u64	dmabuf_destroy	[out] number of dma_buf objects destroyed. When in the idle state, dmabuf_create and dmabuf_destroy counter values are identical

7.14.2.2 struct vion_stats_vrpc

Definition at line 201 of file vion_stats_uapi.h.

Data Fields

__u64	open	[out] number of vRPC open requests: • received if VION_STATS_VRPC_BE stats are requested, • transmitted if VION_STATS_VRPC_FE stats are requested, the preceding statement is true for all the following vRPC requests
__u64	close	[out] number of vRPC close requests. When in the idle state, close and open counter values are identical
__u64	import	[out] number of vRPC import requests: in case the importer client is not trusted or the strict import semantic is enforced, there are as many vRPC import messages transmitted to a peer as importation performed on dma_buf exported by this peer. in case the importer client is trusted and the strict import semantic is relaxed, only the first dma_buf import leads to a vRPC message transmission, hence there may be less vRPC import messages transmitted then importation requests.
__u64	release	[out] number of vRPC release requests. When in the idle state, import and release counter values are identical

7.14.2.3 union vion_stats_res

Definition at line 214 of file vion_stats_uapi.h.

7.14.2.4 struct vion_stats_arg

Definition at line 223 of file vion_stats_uapi.h.

Data Fields

__u32	type	[in] the type of stats to retrieve in enum vion_stats_type range
__u32	peer	[in] peer domain id of the vlink whose stats are retrieved

7.14.2.5 union vion_stats_data

Definition at line 227 of file vion_stats_uapi.h.

7.14.3 Macro Definition Documentation

7.14.3.1 VION_IOC_STATS

```
#define VION_IOC_STATS _IOWR(VION_IOC_MAGIC, 55, union vion_stats_data)
```

Driver statistics command.

This command returns statistics counters maintained by the vION driver. This enables a vION client to determine how many objects have been created and destroyed by the vION driver since its initialization.

Definition at line 184 of file vion_stats_uapi.h.

7.14.3.2 VION_IOC_MAGIC

```
#define VION_IOC_MAGIC ION_IOC_MAGIC
```

driver commands's magic number

This statement provides the "magic number" of vION's ioctl commands. Commands that do not match this number are rejected by the vION driver. Note that vION re-uses the same number as the native ION driver.

Definition at line 129 of file vion_uapi.h.

7.14.4 Enumeration Type Documentation

7.14.4.1 vion_stats_type

```
enum vion_stats_type
```

Enumerator

VION_STATS_DEV	retrieve device related stats
VION_STATS_VRPC_BE	retrieve vRPC back-end (exporter) related stats
VION_STATS_VRPC_FE	retrieve vRPC front-end (importer) related stats

Definition at line 218 of file vion_stats_uapi.h.

7.15 vION Userspace API Introduction

7.15.1 Introduction

vION enhances the functionalities offered by the ION driver, a generalized memory manager originally introduced by Google as part of Android.

Specifically, the vION driver provides in-kernel support for the secure sharing of ION buffers between userspace processes. In the context of virtualized systems, vION extends this concept to embrace the sharing of ION buffers between guest operating systems that execute in separate virtual machines, or domains.

The vion_uapi.h header defines the programming interface exposed by the vION driver to userspace clients.

7.15.2 Background

The ION memory allocator enables userspace clients to allocate buffers from memory regions presented as heaps. Some of these heaps are pre-reserved at boot time while others merely leverage the kernel's standard memory allocators. Each type of device can be provisioned with a different set of standard and platform-specific ION heaps, according to the memory requirements of the device.

Through the API exposed by the /dev/ion device, ION is essentially a service for client processes to allocate `DMA←BUF` buffers that use ION memory heaps as their backing storage.

7.15.3 Features

vION implements the sharing of ION buffers through a secured userspace API that offers buffer export and buffer import commands. The same API enables to transparently share ION buffers between processes that are either hosted by the same guest operating system or by different guests executing in separate domains.

vION does not arbitrate concurrent accesses to the memory exposed by a shared ION buffer. Such concurrent acccesses are expected to be policed by some external token management or producer/consumer service. Similarly, vION does not perform by itself any synchronization operation on shared memory. Processes that share a ION buffer shall use the msync() system call, or DMABUF's finer-grained DMA_BUF_IOCTL_SYNC command to ensure that they always access the buffer's content in a consistent manner.

vION introduces the notion of global buffer identifiers (GIDs) to identify the ION buffers that are exported to other processes. vION's system-wide GIDs uniquely identify ION buffers regardless of the domain it was allocated in.

The API exposed by the ION driver can be secured by the optional use of buffer credentials. Credentials are opaque data structures that match global buffer identifiers. Buffer credentials consists in an 16-byte UUID set to a random value at the buffer's export time. When vION is set to operate in secure mode, vION clients are required to present these credentials in addition to global identifiers to import ION buffers.

7.16 VLX VirtIO Kernel API

Chapters

- VLX VirtIO Kernel Control Plane API
- VLX VirtIO Kernel Data Plane API

7.16.1 Description

The VLX VirtIO kernel API enables implementing VirtIO devices that run on top of the Redbend Hypervisor. These devices consist in back-end drivers that use this API to implement a VirtIO device control and data planes entirely in the Linux kernel.

The configuration of VirtIO devices and drivers with the RedBend Hypervisor is based on device trees. In order to setup a fully functional VirtIO device, the following steps are required.

Front-end driver configuration.

A node which describes the VirtIO device needs to be added to the front-end guest OS device tree. This node makes the guest aware of the device existence and triggers the front-end driver initialization.

```
* virtio_console@3000 {
*     compatible = "virtio,mmio";
*     reg = <0 0x3000 0x160>;
*     interrupts = <0 42 0>;
* };
*
```

Note that this node is not specific to the RedBend Hypervisor and uses the standard Linux device tree binding for VirtIO MMIO devices.

Front-end virtual platform configuration.

The second step is to instruct the hypervisor that a VirtIO MMIO device should be made available in the guest address space. This requires adding another node to the front-end guest OS virtual platform device tree.

The hypervisor uses these information to intercept any accesses on the VirtIO device and forward them to the back-end driver.

```
* virtio_console@3000 {
*     compatible = "vl,virtio-device,mmio";
*     reg = <0 0x3000 0x160>;
*     interrupts = <0 42 0>;
*     vl,device-vmid = <2>;
*     vl,device-name = "console,ivi";
* };
*
```

The compatible property must be set to `vl,virtio-device,mmio` and other properties are defined as follow:

- `reg`: The base address and size of the VirtIO MMIO device in the guest address space.
- `interrupts`: The interrupt that will be delivered to the guest OS when the VirtIO device notifies the front-end driver.
- `vl,device-vmid`: Identify the VM where the back-end driver runs.

- `vl,device-name`: Associate a name to this VirtIO device to pair the device and back-end driver together.

Back-end driver configuration.

The last step is to add a node to the back-end guest OS device tree. It is used to start a specific instance of a VirtIO back-end driver.

```
* ivi_virtio_console {
*     compatible = "vl,virtio-device,console";
*     vl,driver-vmid = <3>;
*     vl,device-name = "console,ivi";
*     vl,uart = "/dev/ttyAMA1";
* };
*
```

The compatible string must match what the back-end driver expects.

The other properties that must be included in such a node are used to indentify a specific instance of a VirtIO device (see vlx_virtio_lookup_device):

- `vl,driver-vmid`: Identify the VM where the front-end driver runs.
- `vl,device-name`: The device name that was specified in the front-end virtual platform definition as part of the previous step.

Note that the `vl,uart` property is an example showing how device tree nodes can be used to pass configuration information to a back-end driver. Different types of back-end driver may require different kind of information.

7.17 VLX VirtIO Kernel Control Plane API

Files

- file vlx-virtio.h

  VLX VirtIO Control Plane API.

Data Structures

- struct vlx_virtio_dev

  Representation of a VirtIO device. More...
- struct vlx_virtio_dev_cb

  VirtIO device-specific operations.

Enumerations

- enum vlx_virtio_api {
  VLX_VIRTIO_KERNEL,
  VLX_VIRTIO_USER,
  VLX_VIRTIO_VHOST }

Functions

- struct vlx_virtio_dev ∗ vlx_virtio_lookup_device (int vmid, const char ∗dev_name, enum vlx_virtio_api api)

  Lookup and claim a VirtIO device using the front-end VM id and the device name.
- void vlx_virtio_release_device (struct vlx_virtio_dev ∗dev)

  Releases a VirtIO device that was previously claimed through vlx_virtio_lookup_device.
- int vlx_virtio_register_device (struct vlx_virtio_dev ∗dev)

  Register a new device with the framework after initialization.
- void vlx_virtio_unregister_device (struct vlx_virtio_dev ∗dev)

  Tell the VLX VirtIO framework that the back-end driver is no longer willing to manage this device.
- int vlx_virtio_device_init_vqs (struct vlx_virtio_dev ∗dev, unsigned int nbr)

  Allocate and initialize the virtqueue descriptors (see vlx_virtio_dev::vqs) for a device.
- void vlx_virtio_device_destroy_vqs (struct vlx_virtio_dev ∗dev)

  Releases the device virtqueue descriptors (see vlx_virtio_dev::vqs).
- void vlx_virtio_config_changed (struct vlx_virtio_dev ∗dev)

  Notify the driver after changes in the device configuration space.
- u32 vlx_virtio_status_get (struct vlx_virtio_dev ∗dev)

  Get the current device status bits.
- void vlx_virtio_status_set (struct vlx_virtio_dev ∗dev, u32 status)

  Set the current device status bits.

7.17.1 Description

7.17.2 Data Structure Documentation

7.17.2.1 struct vlx_virtio_dev

Representation of a VirtIO device.

A vlx_virtio_dev object is initialized by the back-end driver during initialization and registered with the framework through the vlx_virtio_register_device function.

This object holds all the global information regarding a single VirtIO device and may be instantiated multiple times when a single back-end driver handles multiple devices.

Definition at line 136 of file vlx-virtio.h.

Data Fields

unsigned int	api_version	The version of the VLX VirtIO API implemented by this device.
struct vlx_virtq **	vqs	Virtqueues supported by this device.
u32	vendor_id	Virtio vendor ID presented to the front-end driver.
u32	device_id	Virtio device ID presented to the front-end driver.
u64	features	Features supported by this VirtIO device.
u32	config_space_size	Size of the device-specific configuration space in bytes.
atomic_t	config_gen_counter	Configuration space generation counter.
const struct vlx_virtio_dev_cb *	ops	Implementation of device-specific operations.
void *	private_data	Pointer to private_data for use by the back-end driver.

7.17.3 Enumeration Type Documentation

7.17.3.1 vlx_virtio_api

```
enum vlx_virtio_api
```

The VLX VirtIO framework offers flexibility on where to put the VirtIO control plane and data plane implementation. Each of these component can either run inside the kernel or in user space.

Enumerator

VLX_VIRTIO_KERNEL	Both control and data plane are implemented in kernel.
VLX_VIRTIO_USER	Both control and data plane are implemented in user space.
VLX_VIRTIO_VHOST	Control plane is implemented in user space while data plane is running in the kernel using vhost interface.

Definition at line 266 of file vlx-virtio.h.

7.17.4 Function Documentation

7.17.4.1 vlx_virtio_lookup_device()

```
struct vlx_virtio_dev* vlx_virtio_lookup_device (
            int vmid,
            const char * dev_name,
            enum vlx_virtio_api api )
```

Lookup and claim a VirtIO device using the front-end VM id and the device name.

Parameters

in	*vmid*	The id of the VM where the front-end driver runs.
in	*dev_name*	The name assigned to this VirtIO device in the front-end virtual platform device tree.
in	*api*	Specifies the API used to implement this VirtIO device (see vlx_virtio_api for more details on the available options).

Returns

A valid VirtIO device object (see vlx_virtio_dev) if such a device is found, an error pointer otherwise. Possible error codes are:

- `-ENODEV` no device has be found matching this VM id and device name.
- `-EBUSY` this device has already been claimed by another back-end driver.

Note that a single VirtIO device can only be claimed by a single back-end driver. If another back-end driver tries to claim the same device (using the same `vmid` and `dev_name`), vlx_virtio_lookup_device will return `-EBUSY`. A back-end driver can release a VirtIO device with the vlx_virtio_release_device function.

Definition at line 360 of file vlx-virtio-kernel.c.

7.17.4.2 vlx_virtio_release_device()

```
void vlx_virtio_release_device (
            struct vlx_virtio_dev * dev )
```

Releases a VirtIO device that was previously claimed through vlx_virtio_lookup_device.

Parameters

in	*dev*	Reference to the VirtIO device to release.

Definition at line 396 of file vlx-virtio-kernel.c.

7.17.4.3 vlx_virtio_register_device()

```
int vlx_virtio_register_device (
            struct vlx_virtio_dev * dev )
```

Register a new device with the framework after initialization.

Parameters

in	*dev*	Reference to the VirtIO device.

Note that virtio device operations may be invoked as soon as device registration is completed.

Definition at line 406 of file vlx-virtio-kernel.c.

7.17.4.4 vlx_virtio_unregister_device()

```
void vlx_virtio_unregister_device (
            struct vlx_virtio_dev * dev )
```

Tell the VLX VirtIO framework that the back-end driver is no longer willing to manage this device.

Parameters

in	*dev*	Reference to the VirtIO device.

All virtqueues need to be stopped before unregistering a device. The device operations will no longer be invoked after a call to vlx_virtio_unregister_device.

Definition at line 428 of file vlx-virtio-kernel.c.

7.17.4.5 vlx_virtio_device_init_vqs()

```
int vlx_virtio_device_init_vqs (
            struct vlx_virtio_dev * dev,
            unsigned int nbr )
```

Allocate and initialize the virtqueue descriptors (see vlx_virtio_dev::vqs) for a device.

Parameters

in	*dev*	Reference to the VirtIO device.
in	*nbr*	The maximun number of virtqueue this device has.

Returns

0 if the operation succeeded, a negative error code otherwise.

Definition at line 460 of file vlx-virtio-kernel.c.

7.17.4.6 vlx_virtio_device_destroy_vqs()

```
void vlx_virtio_device_destroy_vqs (
            struct vlx_virtio_dev * dev )
```

Releases the device virtqueue descriptors (see vlx_virtio_dev::vqs).

Parameters

in	*dev*	Reference to the VirtIO device.

Definition at line 501 of file vlx-virtio-kernel.c.

7.17.4.7 vlx_virtio_config_changed()

```
void vlx_virtio_config_changed (
            struct vlx_virtio_dev * dev )
```

Notify the driver after changes in the device configuration space.

Parameters

in	*dev*	Reference to the VirtIO device.

This function increments the generation counter for the configuration space of this device and notify the driver of such changes through an interrupt.

Definition at line 509 of file vlx-virtio-kernel.c.

7.17.4.8 vlx_virtio_status_get()

```
u32 vlx_virtio_status_get (
            struct vlx_virtio_dev * dev )
```

Get the current device status bits.

Parameters

in	*dev*	Reference to the VirtIO device.

Returns

The current device status bits.

Definition at line 516 of file vlx-virtio-kernel.c.

7.17.4.9 vlx_virtio_status_set()

```
void vlx_virtio_status_set (
            struct vlx_virtio_dev * dev,
            u32 status )
```

Set the current device status bits.

Parameters

`in`	*dev*	Reference to the VirtIO device.
`in`	*status*	The new status bits to assign to this device.

7.18 VLX VirtIO Kernel Data Plane API

Files

- file vlx-virtq.h

  VLX VirtIO Data Plane API.

Data Structures

- struct vlx_virtq

  Representation of a virtqueue.

Enumerations

- enum kick_policy {
  DEV_WORKER,
  VQ_WORKER,
  DIRECT_CALL }

Functions

- int vlx_vq_init_access (struct vlx_virtq ∗vq)

  Map the virtqueue descriptors and rings.
- void vlx_vq_stop_access (struct vlx_virtq ∗vq)

  Unmap the virtqueue descriptors and rings.
- bool vlx_vq_ring_ready (struct vlx_virtq ∗vq)

  Check whether the virtqueue is ready for I/O.
- bool vlx_vq_has_descs (struct vlx_virtq ∗vq)

  Check whether a virtqueue has available buffers.
- int vlx_vq_getchain (struct vlx_virtq ∗vq, uint16_t ∗pidx, struct kvec ∗iov, int n_iov, uint16_t ∗flags)

  Get the next available chain of descriptors on a virtqueue and put it into an I/O vector.
- void vlx_vq_retchain (struct vlx_virtq ∗vq)

  Return the last chain returned by vlx_vq_getchain back to the available ring.
- void vlx_vq_relchain (struct vlx_virtq ∗vq, uint16_t idx, uint32_t iolen)

  Return specified chain to the used ring, setting its I/O length to the provided value.
- void vlx_vq_endchains (struct vlx_virtq ∗vq, int used_all_avail)

  Driver has finished processing available chains and calling vlx_vq_relchain on each one.
- void vlx_vq_clear_used_ring_flags (struct vlx_virtq ∗vq)

  Helper function for clearing used ring flags.
- void vlx_vq_set_used_ring_flags (struct vlx_virtq ∗vq)

  Helper function for setting used ring flags.
- void vlx_vq_disable_notification (struct vlx_virtq ∗vq)

  Disable driver notifications when new available buffers are added to a virtqueue.
- bool vlx_vq_enable_notification (struct vlx_virtq ∗vq)

  Enable driver notifications when new buffers are available.

7.18.1 Description

7.18.2 Enumeration Type Documentation

7.18.2.1 kick_policy

```
enum kick_policy
```

The VLX VirtIO framework receives kick notifications on virtqueues as software interrupts delivered by the VirtIO bridge in the hypervisor.

The kick notification callbacks provided by the back-end driver (see vlx_virtq::kick) do not have to be invoked in interrupt context directly upon receiveing these notifications and may be invoked through work queues.

Enumerator

DEV_WORKER	The queue notification callbacks are invoked through a worker thread using one work queue per VirtIO device. This is the default kick handler invocation policy.
VQ_WORKER	The queue notification callbacks are invoked through a worker thread using one work queue per virtqueue.
DIRECT_CALL	The queue notification callbacks are invoked directly from interrupt context.

Definition at line 44 of file vlx-virtq.h.

7.18.3 Function Documentation

7.18.3.1 vlx_vq_init_access()

```
int vlx_vq_init_access (
            struct vlx_virtq * vq )
```

Map the virtqueue descriptors and rings.

Parameters

in	*vq*	Reference to the virtqueue.

Returns

0 if the operation succeeded, a negative error code otherwise.

Definition at line 58 of file vlx-virtq.c.

7.18.3.2 vlx_vq_stop_access()

```
void vlx_vq_stop_access (
            struct vlx_virtq * vq )
```

Unmap the virtqueue descriptors and rings.

Parameters

in	*vq*	Reference to the virtqueue.

Definition at line 99 of file vlx-virtq.c.

7.18.3.3 vlx_vq_ring_ready()

```
bool vlx_vq_ring_ready (
            struct vlx_virtq * vq )
```

Check whether the virtqueue is ready for I/O.

Parameters

in	*vq*	Reference to the virtqueue.

Returns

`true` if the virtqueue is ready, `false` otherwise.

Definition at line 129 of file vlx-virtq.c.

7.18.3.4 vlx_vq_has_descs()

```
bool vlx_vq_has_descs (
            struct vlx_virtq * vq )
```

Check whether a virtqueue has available buffers.

Parameters

in	*vq*	Reference to the virtqueue.

Returns

`true` if there are available buffers, `false` otherwise.

Definition at line 137 of file vlx-virtq.c.

7.18.3.5 vlx_vq_getchain()

```
int vlx_vq_getchain (
            struct vlx_virtq * vq,
            uint16_t * pidx,
            struct kvec * iov,
            int n_iov,
            uint16_t * flags )
```

Get the next available chain of descriptors on a virtqueue and put it into an I/O vector.

Parameters

in	*vq*	Reference to the virtqueue.
out	*pidx*	Descriptor index of the chain head.
out	*iov*	Pointer to I/O vector prepared by caller.
in	*n_iov*	Size of `iov` array.
out	*flags*	Pointer to a `uint16_t` array which will contain flags of each descriptor.

Returns

Number of descriptors in the chain on success or a negative error code.

Definition at line 145 of file vlx-virtq.c.

7.18.3.6 vlx_vq_retchain()

```
void vlx_vq_retchain (
            struct vlx_virtq * vq )
```

Return the last chain returned by vlx_vq_getchain back to the available ring.

Parameters

in	*vq*	Reference to the virtqueue.

Definition at line 154 of file vlx-virtq.c.

7.18.3.7 vlx_vq_relchain()

```
void vlx_vq_relchain (
            struct vlx_virtq * vq,
```

```
            uint16_t idx,
            uint32_t iolen )
```

Return specified chain to the used ring, setting its I/O length to the provided value.

Parameters

in	*vq*	Reference to the virtqueue.
in	*idx*	Descriptor index to add to the used ring, as returned by vlx_vq_getchain.
in	*iolen*	Number of data bytes to be returned to the driver.

Definition at line 162 of file vlx-virtq.c.

7.18.3.8 vlx_vq_endchains()

```
void vlx_vq_endchains (
            struct vlx_virtq * vq,
            int used_all_avail )
```

Driver has finished processing available chains and calling vlx_vq_relchain on each one.

If driver used all the available chains, `used_all_avail` need to be set to `1`.

Parameters

in	*vq*	Reference to the virtqueue.
in	*used_all_avail*	Flag indicating whether the back-end driver used all available chains.

Definition at line 170 of file vlx-virtq.c.

7.18.3.9 vlx_vq_clear_used_ring_flags()

```
void vlx_vq_clear_used_ring_flags (
            struct vlx_virtq * vq )
```

Helper function for clearing used ring flags.

Back-end drivers should always use this helper function to clear used ring flags.

Parameters

in	*vq*	Reference to the virtqueue.

Definition at line 178 of file vlx-virtq.c.

7.18.3.10 vlx_vq_set_used_ring_flags()

```
void vlx_vq_set_used_ring_flags (
            struct vlx_virtq * vq )
```

Helper function for setting used ring flags.

Back-end drivers should always use this helper function to set used ring flags.

Parameters

in	*vq*	Reference to the virtqueue.

Definition at line 186 of file vlx-virtq.c.

7.18.3.11 vlx_vq_disable_notification()

```
void vlx_vq_disable_notification (
            struct vlx_virtq * vq )
```

Disable driver notifications when new available buffers are added to a virtqueue.

Parameters

in	*vq*	Reference to the virtqueue.

This is typically used by the back-end driver when it starts processing available buffers. Disabling front-end driver notification acts as an optimization since the back-end driver should notice any extra buffers added to the available ring during this time.

Definition at line 194 of file vlx-virtq.c.

7.18.3.12 vlx_vq_enable_notification()

```
bool vlx_vq_enable_notification (
            struct vlx_virtq * vq )
```

Enable driver notifications when new buffers are available.

Parameters

in	*vq*	Reference to the virtqueue.

Returns

`true` if extra buffers were added to the available ring while re-enabling notifications.

This should be used by the back-end driver to re-enable notifications that were disabled through a previous call to vlx_vq_disable_notification. This is typically used by the back-end driver when it has done processing available buffers.

Note that new buffers may be added to the available list between the moment where the back-end observe the available ring as empty and the moment where notifications are re-enabled. If this happens, the back-end will never receive any notification for these new buffers. Therefore it is important to check the return value from vlx_vq_enable_notification to detect when this condition arises.

Definition at line 202 of file vlx-virtq.c.

7.19 Virtual DMAHEAP Kernel API

Functions

- int vdmaheap_kapi_client_create (struct vdmaheap_client ∗∗vclientp)

    Create a vDMAHEAP client.
- int vdmaheap_kapi_client_release (struct vdmaheap_client ∗vclient)

    Destroy a vDMAHEAP client.
- int vdmaheap_kapi_export (struct vdmaheap_client ∗vclient, struct dma_buf ∗dma_buf, vdmaheap_buffer_gid_t ∗gidp, uuid_t ∗cred)

    Export an DMAHEAP buffer.
- int vdmaheap_kapi_unexport (struct vdmaheap_client ∗vclient, vdmaheap_buffer_gid_t gid, const uuid_t ∗cred)

    Unexport an DMAHEAP buffer.
- int vdmaheap_kapi_import (struct vdmaheap_client ∗vclient, vdmaheap_buffer_gid_t gid, const uuid_t ∗cred, struct dma_buf ∗∗dma_vbufp)

    Import an DMAHEAP buffer.
- int vdmaheap_kapi_import_multiple (struct vdmaheap_client ∗vclient, unsigned int n, vdmaheap_buffer_gid_t gids[ ], const uuid_t ∗creds[ ], struct dma_buf ∗dmabufs[ ])

    Import multiple DMAHEAP buffers.
- void vdmaheap_kapi_set_client_props (struct vdmaheap_client ∗vclient, struct vdmaheap_client_props ∗props)

    Set client's properties.
- void vdmaheap_kapi_get_client_props (struct vdmaheap_client ∗vclient, struct vdmaheap_client_props ∗props)

    Get client's properties.
- int vdmaheap_kapi_link_state (struct vdmaheap_client ∗client, bool wait, __u32 mask, __u32 ∗value)

    Reports the link state with each of the vdmaheap peer devices.

7.19.1 Description

The export/import/close/unexport sequence
Buffer sharing always follows the same export/import/close/unexport sequence:

- An exporter client exports a buffer located on the exporter's domain through the vdmaheap_kapi_export() primitive. The exporters retrieves a couple (gid, credentials) associated with the exported buffer.

- The exporter client then transmits (through a communication channel which is out of the scope of this document) the couple (gid, credentials) to the importer client.

- The importer client then imports the buffer in its own domain through the vdmaheap_kapi_import() / vdmaheap_kapi_import_multiple() primitives. The importer retrieves a struct dma_buf address it can use as if it was a local data structure. The importer may associate a file descriptor to the dma_buf object through the dma_buf_fd() primitive.

- Once the importer client is done with the struct dma_buf, it releases it using sys_close() primitive if it previously invoked dma_buf_fd() or dma_buf_put() otherwise.

- Once the exporter client is done with the buffer, it releases it through the vdmaheap_kapi_unexport() primitive.

As soon as the primitive vdmaheap_kapi_export() is called, the buffer becomes referenced by vDMAHEAP and hence becomes known to it. As soon as the sequence is completed, the buffer becomes unreferenced by v←DMAHEAP, and hence becomes unknown to it again. If at this stage the same buffer was exported through vdmaheap_kapi_export() again, vDMAHEAP would be unable to know if it is actually the same buffer, or a different buffer with the same address, and hence would process it as a different one.

The import cache

The fact that vDMAHEAP unconditionally "forgets" buffers between consecutive export/import/close/unexport sequences leads to sub-optimal performances in the case the same set of buffers is used repeatedly. The reason for this is that the vDMAHEAP exporter and importer peers exchange lots of unnecessary messages to retrieve the buffer layouts considering they are different across sequences whereas they are actually the same with the same layout. The import cache aims at avoiding, under some conditions, these unnecessary message exchanges, remembering buffers beyond sequences during a grace period of time. When the same buffer is re-exported during this period of time, vDMAHEAP exporter peer recognizes it, and assigns the same gid to it. When the same gid import is requested to vDMAHEAP importer peer, the buffer is also recognized, and the existing layout is re-used instead of requesting a new one to the exporter's side. Once the grace period of time has elapsed, the buffer is forgotten by vDMAHEAP, as it would have been without the cache. The import cache is only available to trusted clients which use the relaxed import semantics (i.e. which have their strict_import_semantic attribute set to false). The import cache is controlled through the requested-grace and accepted-grace attributes attached to the clients. Once those attributes have been properly set in vDMAHEAP for the exporter and the importer clients, vDMAH←EAP automatically and transparently manages import cache without any client participation. The requested-grace attribute defines the value requested by the importer client for the grace period of any import requests it issues. The accepted-grace attribute defines the maximum value accepted by the exporter client for any import request it receives. The grace period of time actually assigned to the import is the result of the negotiation between the importer and the exporter sides (i.e. MIN(requested-grace, accepted-grace)). The grace period of time is the duration the import is maintained alive by vDMAHEAP beyond the export/import/close/unexport sequence. Both requested-grace and accepted-grace attributes are expressed in Jiffies. Upon creation of a client a 0 default value is assigned to the requested-grace attribute, and a -1 value is assigned to the accepted-grace attribute, meaning that the client does not requests the imports it issues to be cached, and accept any value of the grace period for any import request it receives. The strict-import-semantic, requested-grace and accepted-grace client's attributes can be modified through the vdmaheap_kapi_set_client_props() primitive.

7.19.2 Function Documentation

7.19.2.1 vdmaheap_kapi_client_create()

```
int vdmaheap_kapi_client_create (
            struct vdmaheap_client ** vclientp )
```

Create a vDMAHEAP client.

This call creates a new client object which enables to invoke vDMAHEAP's core buffer export and buffer import operations.

Parameters

vclientp	[out] a pointer to the new client object

Returns

0 if the client was successfully created
a negative error code otherwise

Definition at line 35 of file vdmaheap_kapi.c.

7.19.2.2 vdmaheap_kapi_client_release()

```
int vdmaheap_kapi_client_release (
            struct vdmaheap_client * vclient )
```

Destroy a vDMAHEAP client.

This call destroys an existing vDMAHEAP client allocated by vdmaheap_kapi_client_create(). All the exports performed by that client are revoked, meaning that they can't be imported any more. This has no effect on the buffers that already have been imported by local or remote clients.

Parameters

vclient	[in] a pointer to an existing vDMAHEAP client

Returns

0 if the client was successfully destroyed
a negative error code otherwise

Definition at line 58 of file vdmaheap_kapi.c.

7.19.2.3 vdmaheap_kapi_export()

```
int vdmaheap_kapi_export (
            struct vdmaheap_client * vclient,
            struct dma_buf * dma_buf,
            vdmaheap_buffer_gid_t * gidp,
            uuid_t * cred )
```

Export an DMAHEAP buffer.

This call exports an existing DMAHEAP buffer identified by a dma_buf object. This call causes a global buffer identifier (GID) and credentials to be associated to the DMAHEAP buffer. If the buffer is exported again, the same GID is returned, but with different credentials. The dma_buf's ref. count is incremented (by one) so that the caller is guaranteed to own a valid reference until unexportation.

Parameters

vclient	[in] a pointer to an existing vDMAHEAP client
dma_buf	[in] a pointer to an existing DMAHEAP buffer identified by a dma_buf.
cred	[out] a pointer to a UUID data structure where the credentials are copied to
gidp	[out] a pointer to the exported buffer's GID

Returns

0 if the buffer was successfully exported
a negative error code otherwise

Definition at line 68 of file vdmaheap_kapi.c.

7.19.2.4 vdmaheap_kapi_unexport()

```
int vdmaheap_kapi_unexport (
            struct vdmaheap_client * vclient,
            vdmaheap_buffer_gid_t gid,
            const uuid_t * cred )
```

Unexport an DMAHEAP buffer.

This calls unexports, i.e. revokes the export of an existing DMAHEAP buffer identified by a dma_buf object. This call has no effect on the buffers that already have been imported by local or remote clients. The associated dma_buf's ref. count is decremented (by one), and the caller is no longer guaranteed of its existence after returning from the call.

Parameters

vclient	[in] a pointer to an existing vDMAHEAP client
gid	[in] a global buffer identifier returned by vdmaheap_kapi_export()
cred	[in] buffer credentials returned by vdmaheap_kapi_export()

Returns

0 if the buffer was successfully unexported
a negative error code otherwise

Definition at line 86 of file vdmaheap_kapi.c.

7.19.2.5 vdmaheap_kapi_import()

```
int vdmaheap_kapi_import (
            struct vdmaheap_client * vclient,
            vdmaheap_buffer_gid_t gid,
            const uuid_t * cred,
            struct dma_buf ** dma_vbufp )
```

Import an DMAHEAP buffer.

This call imports an existing DMAHEAP buffer. This DMAHEAP buffer shall first be exported by a call to vdmaheap_kapi_export(). The dma_buf's ref. count is incremented (by one) so that the caller is guaranteed to own a valid reference until it is released. The caller is expected to call dma_buf_put() when the returned dma_buf is not used any more.

Parameters

vclient	[in] a pointer to an existing vDMAHEAP client
gid	[in] a global buffer identifier returned by vdmaheap_kapi_export()
cred	[in] buffer credentials returned by vdmaheap_kapi_export()
dma_vbufp	[out] a pointer to the dma_buf object of the imported buffer.

Returns

0 if the buffer was successfully imported
a negative error code otherwise

Definition at line 95 of file vdmaheap_kapi.c.

7.19.2.6 vdmaheap_kapi_import_multiple()

```
int vdmaheap_kapi_import_multiple (
            struct vdmaheap_client * vclient,
            unsigned int n,
            vdmaheap_buffer_gid_t gids[],
            const uuid_t * creds[],
            struct dma_buf * dmabufs[] )
```

Import multiple DMAHEAP buffers.

This call imports a list of existing DMAHEAP buffers. These DMAHEAP buffers shall first be exported by a call to vdmaheap_kapi_export(). The dma_buf's ref. counts are incremented (by one) so that the caller is guaranteed to own valid references on buffers until they are released. The caller is expected to call dma_buf_put() when a returned dma_buf is not used any more. This call is optimized for the case where all buffers are originated (i.e. were exported) from a single domain. VDMAHEAP will perform the import request using the least possible number of vRPC transactions in accordance to the available amount of pmem. In the best case (i.e. if all imported buffers layout fit in the vDMAHEAP pmem), vDMAHEAP will perform a single vRPC transaction. In the worst case (i.e. if a single buffer layout fits in the vDMAHEAP pmem), vDMAHEAP will perform as many vRPC transactions as imported buffers. Note that vDMAHEAP pmem must be sized to host an entire buffer layout at least.

Parameters

vclient	[in] a pointer to an existing vDMAHEAP client.
n	[in] the number of buffers to be imported without limitation.
gids	[in] an array of global buffer identifiers returned by vdmaheap_kapi_export().
creds	[in] an array of buffer credentials returned by vdmaheap_kapi_export(), or NULL if credentials are not provided. if non-NULL, this array must be large enough to contain the n requested credential pointers.
dmabufs	[out] an array of pointers to the dma_buf objects of the imported buffers, or ERR_PTR() if a buffer failed to be imported. This array must be large enough to contain the n requested dma_buf.

Returns

0 if all the buffers were successfully imported
a negative error code otherwise. In that case, the dmabufs array shall be parsed to know which buffers failed to be imported.

Definition at line 114 of file vdmaheap_kapi.c.

7.19.2.7 vdmaheap_kapi_set_client_props()

```
void vdmaheap_kapi_set_client_props (
            struct vdmaheap_client * vclient,
            struct vdmaheap_client_props * props )
```

Set client's properties.

This call sets client's properties.

Parameters

vclient	[in] a pointer to an existing vDMAHEAP client
props	[in] a pointer to a property data structure whose values will be set in the client.

Returns

nothing

Definition at line 122 of file vdmaheap_kapi.c.

7.19.2.8 vdmaheap_kapi_get_client_props()

```
void vdmaheap_kapi_get_client_props (
            struct vdmaheap_client * vclient,
            struct vdmaheap_client_props * props )
```

Get client's properties.

This call gets client's properties.

Parameters

vclient	[in] a pointer to an existing vDMAHEAP client
props	[out] a pointer to a property data structure where properties values will be copied.

Returns

nothing

Definition at line 131 of file vdmaheap_kapi.c.

7.19.2.9 vdmaheap_kapi_link_state()

```
int vdmaheap_kapi_link_state (
            struct vdmaheap_client * client,
            bool wait,
            __u32 mask,
            __u32 * value )
```

Reports the link state with each of the vdmaheap peer devices.

The link state with vdmaheap peer devices is returned in a 32 bit bitmap where bit i indicates the link state with vdmaheap device located on VMi. The function can run in 2 modes depending on the wait parameter:

- wait is false: returns the current link state in the location pointed by value,
- wait is true: blocks until the link state with peers selected by mask equals the value pointed by value. Eventually, the location pointed by value is updated with the last read link state.

Parameters

client	[in] a pointer to an existing vDMAHEAP client
wait	[in] mode the function operates in.
mask	[in] bitmap of the peer devices to which link state is to be monitored,
value	[in/out] bitmap of the link states to be monitored.

Returns

0 if the link state is successfully returned
a negative error code otherwise.

Definition at line 140 of file vdmaheap_kapi.c.

7.20 Virtual DMA fence v2 Kernel API

Files

- file vfence2_kapi.h

  virtual DMA fence v2 (vFence2) driver - kernel mode API

Functions

- int vfence2_create_client (const char *name, struct vfence2_client **vfcl) __must_check

  Create a vfence client.
- int vfence2_destroy_client (struct vfence2_client *vfcl)

  Destroy a vfence client.
- int vfence2_connect (struct vfence2_client *vfcl, uint32_t vmid, struct vfence2_connection **vfco) __must↩
_check

  Connect to a peer VM.
- int vfence2_disconnect (struct vfence2_connection *vfco)

  Disconnect a peer VM, previous connected with vfence2_connect()
- int vfence2_export_fence (struct vfence2_connection *vfco, int fd, vfence2_id_t *gid, bool can_block) __↩
must_check

  Export a DMA fence.
- int vfence2_unexport_fence (struct vfence2_connection *vfco, vfence2_id_t gid)

  Unexport a DMA fence.
- int vfence2_import_fence (struct vfence2_connection *vfco, vfence2_id_t gid, int *fd) __must_check

  Import a DMA fence.

7.20.1 Description

Objective of the API:

- Facilitate dma_fence virtualisation by offering virtual drivers a shared kernel module exporting an API.
- This kernel module can additionally manage a user mode API.

The export/import/close sequence
DMA fence sharing always follows the same export/import/close sequence:

- An exporter client exports a DMA fence located in the exporter's domain through the vfence2_export_fence() primitive. The exporter retrieves a vfence2_id_t associated with the exported DMA fence.
- The exporter client then transmits (through a communication channel which is out of the scope of this document) the vfence2_id_t to the importer client.
- The importer client then imports the DMA fence into its own domain through the vfence2_import_fence() primitive. The importer retrieves a local file descriptor.
- Once the importer client is done with the file, it releases it using sys_close() primitive.

7.20.2 Function Documentation

7.20.2.1 vfence2_create_client()

```
int vfence2_create_client (
            const char * name,
            struct vfence2_client ** vfcl )
```

Create a vfence client.

Parameters

name	[in] Name used to distinguish clients. It is remembered by reference until vfence2_destroy_client().
vfcl	[out] Pointer to the newly created client object

Returns

0 on success
a negative error code otherwise

This call creates a new client object which enables to invoke DMA fence export and import operations. The call is non-blocking.

Definition at line 4338 of file vfence2.c.

7.20.2.2 vfence2_destroy_client()

```
int vfence2_destroy_client (
            struct vfence2_client * vfcl )
```

Destroy a vfence client.

Parameters

vfcl	[in] vfence client descriptor obtained from vfence2_create_client()

Returns

0 if the client was successfully destroyed
a negative error code otherwise

This call destroys an existing vfence client allocated by vfence2_create_client(). The call is non-blocking. All connections must have been disconnected beforehand, or -EBUSY will be returned.

Definition at line 4384 of file vfence2.c.

7.20.2.3 vfence2_connect()

```
int vfence2_connect (
            struct vfence2_client * vfcl,
            uint32_t vmid,
            struct vfence2_connection ** vfco )
```

Connect to a peer VM.

Parameters

vfcl	[in] vfence client
vmid	[in] Peer VM to communicate with
vfco	[out] Returned struct vfence2_connection descriptor

Returns

0 on success
a negative error code otherwise

The call is non-blocking and merely confirms the existence of a connection to a given peer VM. The same connection can be used for exporting and for importing. Several connections can be opened by each client. Connection cannot be towards the local VM.

Definition at line 4752 of file vfence2.c.

7.20.2.4 vfence2_disconnect()

```
int vfence2_disconnect (
            struct vfence2_connection * vfco )
```

Disconnect a peer VM, previous connected with vfence2_connect()

Parameters

vfco	[in] Connection to disconnect

Returns

0 on success
a negative error code otherwise

The call is non-blocking. It invalidates the passed descriptor. All the exports performed on that connection are revoked, meaning that they can't be imported any more (except if another connection also exported them). This has no effect on the DMA fences that already have been imported.

Definition at line 4765 of file vfence2.c.

7.20.2.5 vfence2_export_fence()

```
int vfence2_export_fence (
            struct vfence2_connection * vfco,
            int fd,
            vfence2_id_t * gid,
            bool can_block )
```

Export a DMA fence.

Parameters

vfco	[in] Open vfence connection
fd	[in] DMA fence file descriptor to export
gid	[out] Returned vfence identifier, or zero, if vfence2 driver has successfully waited for the fence locally
can_block	Whether blocking is allowed in situations where the peer VM is not up

Returns

0 on success
-EAGAIN blocking required but can_block is false
another negative error code otherwise

If the peer VM is up, this call exports a DMA fence identified by the passed file descriptor. It causes a vfence identifier (vfence2_id_t) to be associated with the DMA fence. If the DMA fence is exported again, the same vfence identifier might be returned. In this case, a reference count is incremented each time so that the every caller is guaranteed to own a valid reference. If can_block is set, and the peer VM is not up, this call will wait for fence locally. If the DMA fence has been successfully waited for, the call succeeds but a zero gid (VFENCE2_ID_NONE) is returned.

Definition at line 4801 of file vfence2.c.

7.20.2.6 vfence2_unexport_fence()

```
int vfence2_unexport_fence (
            struct vfence2_connection * vfco,
            vfence2_id_t gid )
```

Unexport a DMA fence.

Parameters

vfco	[in] Open vfence connection
gid	[in] vfence identifier

Returns

0 if the DMA fence was successfully unexported
a negative error code otherwise

This calls unexports, i.e. revokes the export of an existing DMA fence identified by a file descriptor. It is only supposed to be used in error handling paths. This call may signal DMA fences which have already been imported. The associated file descriptor reference count is decremented, and the caller is no longer guaranteed of its validity after return from the call. The call is non-blocking.

Definition at line 4888 of file vfence2.c.

7.20.2.7 vfence2_import_fence()

```
int vfence2_import_fence (
            struct vfence2_connection * vfco,
            vfence2_id_t gid,
            int * fd )
```

Import a DMA fence.

Parameters

vfco	[in] Open vfence connection
gid	[in] vfence identifier returned by vfence2_export_fence()
fd	[out] DMA fence file descriptor

Returns

0 on success
a negative error code otherwise

This call imports an existing DMA fence given a valid (non-zero) gid. This DMA fence shall first be exported by a call to vfence2_export_fence(). A gid returned by a vfence2_export_fence() call can only be imported once. Additional calls with same gid will error, except if the same gid has again been returned by another vfence2_export_fence() call. Caller is expected to close the file descriptor using sys_close() when it is not necessary any more. The call is non-blocking.

Definition at line 5040 of file vfence2.c.

7.21 Virtual ION Kernel API

Functions

- int vion_kapi_client_create (struct vion_client **vclientp)

  Create a vION client.
- int vion_kapi_client_release (struct vion_client *vclient)

  Destroy a vION client.
- int vion_kapi_export (struct vion_client *vclient, struct dma_buf *dma_buf, vion_buffer_gid_t *gidp, uuid_t *cred)

  Export an ION buffer.
- int vion_kapi_unexport (struct vion_client *vclient, vion_buffer_gid_t gid, const uuid_t *cred)

  Unexport an ION buffer.
- int vion_kapi_import (struct vion_client *vclient, vion_buffer_gid_t gid, const uuid_t *cred, struct dma_buf **dma_vbufp)

  Import an ION buffer.
- int vion_kapi_import_multiple (struct vion_client *vclient, unsigned int n, vion_buffer_gid_t gids[ ], const uuid_t *creds[ ], struct dma_buf *dmabufs[ ])

  Import multiple ION buffers.
- void vion_kapi_set_client_props (struct vion_client *vclient, struct vion_client_props *props)

  Set client's properties.
- void vion_kapi_get_client_props (struct vion_client *vclient, struct vion_client_props *props)

  Get client's properties.

7.21.1 Description

The export/import/close/unexport sequence
Buffer sharing always follows the same export/import/close/unexport sequence:

- An exporter client exports a buffer located on the exporter's domain through the vion_kapi_export() primitive. The exporters retrieves a couple (gid, credentials) associated with the exported buffer.
- The exporter client then transmits (through a communication channel which is out of the scope of this document) the couple (gid, credentials) to the importer client.
- The importer client then imports the buffer in its own domain through the vion_kapi_import() / vion_kapi_import_multiple() primitives. The importer retrieves a struct dma_buf address it can use as if it was a local data structure. The importer may associate a file descriptor to the dma_buf object through the dma_buf_fd() primitive.
- Once the importer client is done with the struct dma_buf, it releases it using sys_close() primitive if it previously invoked dma_buf_fd() or dma_buf_put() otherwise.
- Once the exporter client is done with the buffer, it releases it through the vion_kapi_unexport() primitive.

As soon as the primitive vion_kapi_export() is called, the buffer becomes referenced by vION and hence becomes known to it. As soon as the sequence is completed, the buffer becomes unreferenced by vION, and hence becomes unknown to it again. If at this stage the same buffer was exported through vion_kapi_export() again, vION would be unable to know if it is actually the same buffer, or a different buffer with the same address, and hence would process it as a different one.

The import cache
The fact that vION unconditionally "forgets" buffers between consecutive export/import/close/unexport sequences leads to sub-optimal performances in the case the same set of buffers is used repeatedly. The reason for this is that the vION exporter and importer peers exchange lots of unnecessary messages to retrieve the buffer layouts considering they are different across sequences whereas they are actually the same with the same layout. The import cache aims at avoiding, under some conditions, these unnecessary message exchanges, remembering buffers beyond sequences during a grace period of time. When the same buffer is re-exported during this period of time, vION exporter peer recognizes it, and assigns the same gid to it. When the same gid import is requested to vION importer peer, the buffer is also recognized, and the existing layout is re-used instead of requesting a new one to the exporter's side. Once the grace period of time has elapsed, the buffer is forgotten by vION, as it would have been without the cache. The import cache is only available to trusted clients which use the relaxed import semantics (i.e. which have their strict_import_semantic attribute set to false). The import cache is controlled through the requested-grace and accepted-grace attributes attached to the clients. Once those attributes have been properly set in vION for the exporter and the importer clients, vION automatically and transparently manages import cache without any client participation. The requested-grace attribute defines the value requested by the importer client for the grace period of any import requests it issues. The accepted-grace attribute defines the maximum value accepted by the exporter client for any import request it receives. The grace period of time actually assigned to the import is the result of the negotiation between the importer and the exporter sides (i.e. MIN(requested-grace, accepted-grace)). The grace period of time is the duration the import is maintained alive by vION beyond the export/import/close/unexport sequence. Both requested-grace and accepted-grace attributes are expressed in Jiffies. Upon creation of a client a 0 default value is assigned to the requested-grace attribute, and a -1 value is assigned to the accepted-grace attribute, meaning that the client does not requests the imports it issues to be cached, and accept any value of the grace period for any import request it receives. The strict-import-semantic, requested-grace and accepted-grace client's attributes can be modified through the vion_kapi_set_client_props() primitive.

7.21.2 Function Documentation

7.21.2.1 vion_kapi_client_create()

```
int vion_kapi_client_create (
            struct vion_client ** vclientp )
```

Create a vION client.

This call creates a new client object which enables to invoke vION's core buffer export and buffer import operations.

Parameters

vclientp	[out] a pointer to the new client object

Returns

0 if the client was successfully created
a negative error code otherwise

Definition at line 35 of file vion_kapi.c.

7.21.2.2 vion_kapi_client_release()

```
int vion_kapi_client_release (
            struct vion_client * vclient )
```

Destroy a vION client.

This call destroys an existing vION client allocated by vion_kapi_client_create(). All the exports performed by that client are revoked, meaning that they can't be imported any more. This has no effect on the buffers that already have been imported by local or remote clients.

Parameters

vclient	[in] a pointer to an existing vION client

Returns

0 if the client was successfully destroyed
a negative error code otherwise

Definition at line 58 of file vion_kapi.c.

7.21.2.3 vion_kapi_export()

```
int vion_kapi_export (
            struct vion_client * vclient,
            struct dma_buf * dma_buf,
            vion_buffer_gid_t * gidp,
            uuid_t * cred )
```

Export an ION buffer.

This call exports an existing ION buffer identified by a dma_buf object. This call causes a global buffer identifier (GID) and credentials to be associated to the ION buffer. If the buffer is exported again, the same GID is returned, but with different credentials. The dma_buf's ref. count is incremented (by one) so that the caller is guaranteed to own a valid reference until unexportation.

Parameters

vclient	[in] a pointer to an existing vION client
dma_buf	[in] a pointer to an existing ION buffer identified by a dma_buf.
cred	[out] a pointer to a UUID data structure where the credentials are copied to
gidp	[out] a pointer to the exported buffer's GID

Returns

0 if the buffer was successfully exported
a negative error code otherwise

Definition at line 68 of file vion_kapi.c.

7.21.2.4 vion_kapi_unexport()

```
int vion_kapi_unexport (
            struct vion_client * vclient,
            vion_buffer_gid_t gid,
            const uuid_t * cred )
```

Unexport an ION buffer.

This calls unexports, i.e. revokes the export of an existing ION buffer identified by a dma_buf object. This call has no effect on the buffers that already have been imported by local or remote clients. The associated dma_buf's ref. count is decremented (by one), and the caller is no longer guaranteed of its existence after returning from the call.

Parameters

vclient	[in] a pointer to an existing vION client
gid	[in] a global buffer identifier returned by vion_kapi_export()
cred	[in] buffer credentials returned by vion_kapi_export()

Returns

0 if the buffer was successfully unexported
a negative error code otherwise

Definition at line 86 of file vion_kapi.c.

7.21.2.5 vion_kapi_import()

```
int vion_kapi_import (
            struct vion_client * vclient,
            vion_buffer_gid_t gid,
            const uuid_t * cred,
            struct dma_buf ** dma_vbufp )
```

Import an ION buffer.

This call imports an existing ION buffer. This ION buffer shall first be exported by a call to vion_kapi_export(). The dma_buf's ref. count is incremented (by one) so that the caller is guaranteed to own a valid reference until it is released. The caller is expected to call dma_buf_put() when the returned dma_buf is not used any more.

Parameters

vclient	[in] a pointer to an existing vION client
gid	[in] a global buffer identifier returned by vion_kapi_export()
cred	[in] buffer credentials returned by vion_kapi_export()
dma_vbufp	[out] a pointer to the dma_buf object of the imported buffer.

Returns

0 if the buffer was successfully imported
a negative error code otherwise

Definition at line 95 of file vion_kapi.c.

7.21.2.6 vion_kapi_import_multiple()

```
int vion_kapi_import_multiple (
            struct vion_client * vclient,
            unsigned int n,
            vion_buffer_gid_t gids[ ],
            const uuid_t * creds[ ],
            struct dma_buf * dmabufs[ ] )
```

Import multiple ION buffers.

This call imports a list of existing ION buffers. These ION buffers shall first be exported by a call to vion_kapi_export(). The dma_buf's ref. counts are incremented (by one) so that the caller is guaranteed to own valid references on buffers until they are released. The caller is expected to call dma_buf_put() when a returned dma_buf is not used any more. This call is optimized for the case where all buffers are originated (i.e. were exported) from a single domain. VION will perform the import request using the least possible number of vRPC transactions in accordance to the available amount of pmem. In the best case (i.e. if all imported buffers layout fit in the vION pmem), vION will perform a single vRPC transaction. In the worst case (i.e. if a single buffer layout fits in the vION pmem), vION will perform as many vRPC transactions as imported buffers. Note that vION pmem must be sized to host an entire buffer layout at least.

Parameters

vclient	[in] a pointer to an existing vION client.
n	[in] the number of buffers to be imported without limitation.
gids	[in] an array of global buffer identifiers returned by vion_kapi_export().
creds	[in] an array of buffer credentials returned by vion_kapi_export(), or NULL if credentials are not provided. if non-NULL, this array must be large enough to contain the n requested credential pointers.
dmabufs	[out] an array of pointers to the dma_buf objects of the imported buffers, or ERR_PTR() if a buffer failed to be imported. This array must be large enough to contain the n requested dma_buf.

Returns

0 if all the buffers were successfully imported
a negative error code otherwise. In that case, the dmabufs array shall be parsed to know which buffers failed to be imported.

Definition at line 114 of file vion_kapi.c.

7.21.2.7 vion_kapi_set_client_props()

```
void vion_kapi_set_client_props (
            struct vion_client * vclient,
            struct vion_client_props * props )
```

Set client's properties.

This call sets client's properties.

Parameters

vclient	[in] a pointer to an existing vION client
props	[in] a pointer to a property data structure whose values will be set in the client.

Returns

nothing

Definition at line 122 of file vion_kapi.c.

7.21.2.8 vion_kapi_get_client_props()

```
void vion_kapi_get_client_props (
            struct vion_client * vclient,
            struct vion_client_props * props )
```

Get client's properties.

This call gets client's properties.

Parameters

vclient	[in] a pointer to an existing vION client
props	[out] a pointer to a property data structure where properties values will be copied.

Returns

nothing

Definition at line 131 of file vion_kapi.c.

7.22 VGKI kernel-mode API

Data Structures

- struct vgki_thread

  Thread library allowing clean termination of VGKI threads.

Functions

- struct file * vgki_kapi_filp_open (const char *path, int flags, int mode)

  Replacement for `filp_open()`. Open file using arguments similar to `open(2)` ones.
- pid_t vgki_kapi_kernel_thread (int(*thread)(void *), void *data, int flags)

  Replacement for `kernel_thread()`. Create a thread with richer semantics than `kthread_create()` ones.
- int vgki_kapi_open (const char *path, int flags, int mode)

  Replacement for `open(2)`.
- int vgki_kapi_close (int fd)

  Replacement for `close(2)`.
- void __user * vgki_kapi_mmap (unsigned long phys_addr, size_t size)

  Indirect replacement for `set_fs()`. Allows to create a user-mode mapping which can be passed to kernel functions expecting user-mode buffers. This mapping is only valid inside VGKI threads.
- int vgki_kapi_munmap (void __user *addr, size_t size)

  Opposite of `vgki_kapi_mmap()`. Removes the user-mode mapping.
- static pid_t vgki_thread_pid (const struct vgki_thread *thread)

  Extract thread identifier from `vgki_thread` object. Indeed, `vgki_thread_start()` does not return the PID.
- long vgki_thread_join (struct vgki_thread *thread)

  Wait until `vgki_thread` exits from start function.
- int vgki_thread_start (struct vgki_thread *thread, int(*func)(void *), void *data, const char *name)

  Start a `vgki_thread`.

struct vgki_thread

- int(* vgki_thread::func)(void *data)

  Thread start routine.
- void * vgki_thread::data

  Argument to start function.
- const char * vgki_thread::name

  Name for the thread.
- pid_t vgki_thread::pid

  Thread identifier of the thread.
- struct completion vgki_thread::completion

  Object allowing to safely await termination of thread.
- long vgki_thread::exit_code

  Exit value of the thread start routine.

7.22.1 Description

7.22.2 Cross References

Related Documents
Component Interface
Manual Page

7.22.3 Function Documentation

7.22.3.1 vgki_kapi_filp_open()

```
struct file * vgki_kapi_filp_open (
            const char * path,
            int flags,
            int mode )
```

Replacement for `filp_open()`. Open file using arguments similar to `open(2)` ones.

Parameters

path	Absolute pathname of file to open. No specific current directory should be assumed.
flags	`O_RDWR`, `O_CREAT`, `O_TRUNC`, etc.
mode	File mode bits if `O_CREAT` or `O_TMPFILE` passed among flags

Returns

An open file object pointer (valid in any thread), or error code encoded as pointer using `ERR_PTR()` macro

Definition at line 1214 of file vgki.c.

7.22.3.2 vgki_kapi_kernel_thread()

```
pid_t vgki_kapi_kernel_thread (
            int(*)(void *) thread,
            void * data,
            int flags )
```

Replacement for `kernel_thread()`. Create a thread with richer semantics than `kthread_create()` ones.

Parameters

thread	Start function
data	Argument to start function
flags	`CLONE_FS` | `CLONE_FILES` | ... | `SIGCHLD`

Returns

A process identifier, or negative error code. The PID is not waitable, because caller does not become parent.

Definition at line 1243 of file vgki.c.

7.22.3.3 vgki_kapi_open()

```
int vgki_kapi_open (
            const char * path,
            int flags,
            int mode )
```

Replacement for `open(2)`.

Parameters

path	Absolute pathname of file to open. No specific current directory should be assumed.
flags	`O_RDWR`, `O_CREAT`, `O_TRUNC`, etc.
mode	File mode bits if `O_CREAT` or `O_TMPFILE` passed among flags

Returns

A file descriptor (valid only inside VGKI threads), or negative error code

Definition at line 1271 of file vgki.c.

7.22.3.4 vgki_kapi_close()

```
int vgki_kapi_close (
            int fd )
```

Replacement for `close(2)`.

Parameters

fd	File descriptor obtained from `vgki_kapi_open()` or otherwise allocated from a VGKI thread

Returns

0 on success, or negative error code

Definition at line 1300 of file vgki.c.

7.22.3.5 vgki_kapi_mmap()

```
void __user * vgki_kapi_mmap (
            unsigned long phys_addr,
            size_t size )
```

Indirect replacement for `set_fs()`. Allows to create a user-mode mapping which can be passed to kernel functions expecting user-mode buffers. This mapping is only valid inside VGKI threads.

Parameters

phys_addr	Page-aligned physical address of a kernel-mode data object. This object must be contiguous in physical memory, for example located in PDEV or PMEM memory, or allocated with `__get_free_pages()`.
size	Page-aligned size of the kernel-mode data object.

Returns

User-mode virtual address inside the `vgki-helper` process, or error code encoded as pointer using `ERR_PTR()` macro.

Definition at line 1326 of file vgki.c.

7.22.3.6 vgki_kapi_munmap()

```
int vgki_kapi_munmap (
            void __user * addr,
            size_t size )
```

Opposite of `vgki_kapi_mmap()`. Removes the user-mode mapping.

Parameters

addr	Page-aligned user-mode virtual address previously obtained from `vgki_kapi_mmap()`
size	Page-aligned size of the kernel-mode data object previously passed to `vgki_kapi_mmap()`

Returns

0 on success or negative error code

7.22.3.7 vgki_thread_pid()

```
pid_t vgki_thread_pid (
            const struct vgki_thread * thread )  [inline], [static]
```

Extract thread identifier from `vgki_thread` object. Indeed, `vgki_thread_start()` does not return the PID.

Parameters

thread	A vgki_thread object pointer

Returns

pid thread identifier

Definition at line 51 of file vgki-kapi.h.

7.22.3.8 vgki_thread_join()

```
long vgki_thread_join (
            struct vgki_thread * thread )
```

Wait until vgki_thread exits from start function.

Parameters

thread	A vgki_thread object pointer

Returns

thread exit code

Definition at line 1401 of file vgki.c.

7.22.3.9 vgki_thread_start()

```
int vgki_thread_start (
            struct vgki_thread * thread,
            int(*)(void *) func,
            void * data,
            const char * name )
```

Start a vgki_thread.

Parameters

thread	A vgki_thread object pointer
func	Start function
data	Argument to start function
name	Name for the thread

Returns

0 on success or negative error code. The PID can be retrieved using `vgki_thread_pid()`

Definition at line 1416 of file vgki.c.

Chapter 8

Data Structure Documentation

8.1 .be Struct Reference

8.1.1 Detailed Description

Definition at line 229 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.2 .be Struct Reference

8.2.1 Detailed Description

Definition at line 229 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.3 .be Struct Reference

8.3.1 Detailed Description

Definition at line 229 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.4 .be Struct Reference

8.4.1 Detailed Description

Definition at line 229 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.5 .be Struct Reference

8.5.1 Detailed Description

Definition at line 229 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.6 .user Struct Reference

8.6.1 Detailed Description

Definition at line 212 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.7 .user.ioctl Struct Reference

8.7.1 Detailed Description

Definition at line 216 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.8 _cdata_header Union Reference

8.8.1 Detailed Description

Definition at line 57 of file vlx-bootloader-control.c.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-bootloader-control.c

8.9 _cdata_header.__unnamed__ Struct Reference

8.9.1 Detailed Description

Definition at line 59 of file vlx-bootloader-control.c.

The documentation for this struct was generated from the following files:

8.10 _proc_file_t Struct Reference

8.10.1 Detailed Description

Definition at line 546 of file htf-be-ui-procfs.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/htf/htf-be-ui-procfs.c

8.11 _s2mpu_sysconf_ops Struct Reference

8.11.1 Detailed Description

Definition at line 54 of file s2mpu-fault-info-priv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/s2mpu-fault-info-priv.h

8.12 _vlx_port_t Struct Reference

8.12.1 Detailed Description

Definition at line 51 of file vlx-console.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-console.c

8.13 avg_stats Struct Reference

8.13.1 Detailed Description

Definition at line 134 of file vdmaheap_dev.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_dev.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_dev.h

8.14 backend_t Struct Reference

8.14.1 Detailed Description

Definition at line 284 of file vevdev-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-be.c

8.15 be_test_ui_t Struct Reference

8.15.1 Detailed Description

Definition at line 45 of file htf-be-ui-procfs.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/htf/htf-be-ui-procfs.c

8.16 be_ui_t Struct Reference

8.16.1 Detailed Description

Definition at line 68 of file htf-be-ui-procfs.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/htf/htf-be-ui-procfs.c

8.17 buffer_info Struct Reference

8.17.1 Detailed Description

Definition at line 31 of file vlx-vq-packed.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vq-packed.c

8.18 device_abs_buf_t Struct Reference

8.18.1 Detailed Description

Definition at line 254 of file vevdev-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-be.c

8.19 device_abs_pip_t Struct Reference

8.19.1 Detailed Description

Definition at line 259 of file vevdev-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-be.c

8.20 device_status_t Struct Reference

8.20.1 Detailed Description

Definition at line 250 of file vevdev-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-be.c

8.21 device_t Struct Reference

8.21.1 Detailed Description

Definition at line 342 of file vevdev-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-be.c

8.22 domain_t Struct Reference

8.22.1 Detailed Description

Definition at line 72 of file vlx-pm-domain-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-fe.c

8.23 evt_ring Struct Reference

8.23.1 Detailed Description

Definition at line 42 of file vlx-event-rings.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-rings.c

8.24 evt_rings_dev Struct Reference

8.24.1 Detailed Description

Definition at line 71 of file vlx-event-rings.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-rings.c

8.25 evt_stat_array_t Struct Reference

8.25.1 Detailed Description

Definition at line 49 of file vlx-event-stats.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-stats.c

8.26 evt_stat_dev_t Struct Reference

8.26.1 Detailed Description

Definition at line 100 of file vlx-event-stats.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-stats.c

8.27 evt_stat_pmem_t Struct Reference

8.27.1 Detailed Description

Definition at line 92 of file vlx-event-stats.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-stats.c

8.28 evtlog_dev_t Struct Reference

8.28.1 Detailed Description

Definition at line 71 of file vlx-event-log.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-log.c

8.29 evtlog_pmem_t Struct Reference

8.29.1 Detailed Description

Definition at line 46 of file vlx-event-log.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-log.c

8.30 evtlog_ring_t Struct Reference

8.30.1 Detailed Description

Definition at line 54 of file vlx-event-log.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-log.c

8.31 ExDev Struct Reference

8.31.1 Detailed Description

Definition at line 193 of file vbpipe.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbpipe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vpipe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vupipe.c

8.32 ExRing Struct Reference

8.32.1 Detailed Description

Definition at line 103 of file vbpipe.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbpipe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vpipe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vupipe.c

8.33 fe_ui_t Struct Reference

8.33.1 Detailed Description

Definition at line 54 of file htf-be-ui-procfs.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/htf/htf-be-ui-procfs.c

8.34 filter_entry_desc_t Struct Reference

8.34.1 Detailed Description

Definition at line 104 of file vlx-event-log-filtering.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-log-filtering.c

8.35 filter_mask_operation_t Struct Reference

8.35.1 Detailed Description

Definition at line 76 of file vlx-event-log-filtering.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-log-filtering.c

8.36 frontend_t Struct Reference

8.36.1 Detailed Description

Definition at line 238 of file vevdev-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-be.c

8.37 htf_fe_utest_t Struct Reference

8.37.1 Detailed Description

Definition at line 42 of file htf-fe-ui.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/htf/htf-fe-ui.c

8.38 htf_ioctl_param_t Struct Reference

8.38.1 Detailed Description

Definition at line 87 of file htf-slave-ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/htf-slave-ioctl.h

8.39 htf_work_t Struct Reference

8.39.1 Detailed Description

Definition at line 186 of file htf-osal-sup.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/htf-osal-sup.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/htf-osal-usr.h

8.40 id2tag Struct Reference

8.40.1 Detailed Description

Definition at line 109 of file vwatchdog.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vwatchdog.c

8.41 info_vlinks_ep_t Struct Reference

8.41.1 Detailed Description

Definition at line 71 of file vlx-info-vlinks.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-info-vlinks.c

8.42 input_dev Struct Reference

8.42.1 Detailed Description

Definition at line 28 of file vevdev-expr-idev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev-expr-idev.h

8.43 input_id Struct Reference

8.43.1 Detailed Description

Definition at line 21 of file vevdev-expr-idev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev-expr-idev.h

8.44 input_vevent Struct Reference

8.44.1 Detailed Description

Definition at line 29 of file vevdev_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev_common.h

8.45 last_kmsg_t Struct Reference

8.45.1 Detailed Description

Definition at line 63 of file vlx-last-kmsg.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-last-kmsg.c

8.46 log_buf Struct Reference

8.46.1 Detailed Description

Definition at line 67 of file vlx-log.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-log.c

8.47 log_file Struct Reference

8.47.1 Detailed Description

Definition at line 76 of file vlx-log.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-log.c

8.48 name2id Struct Reference

8.48.1 Detailed Description

Definition at line 82 of file vlx-bootloader-control.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-bootloader-control.c

8.49 NK_DDI_VERSION Struct Reference

8.49.1 Detailed Description

Definition at line 8 of file version.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/version.h

8.50 nk_idle_event_t Struct Reference

8.50.1 Detailed Description

Definition at line 28 of file pm.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pm.h

8.51 nk_idle_info_t Struct Reference

8.51.1 Detailed Description

Definition at line 34 of file pm-hmp.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pm-hmp.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pm-smp.c

8.52 nk_sysconf Struct Reference

8.52.1 Detailed Description

Definition at line 1730 of file ddi.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/hyp/ddi.c

8.53 nk_vlink_item Struct Reference

8.53.1 Detailed Description

Definition at line 1736 of file ddi.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/hyp/ddi.c

8.54 nk_xirq_wrapper Struct Reference

8.54.1 Detailed Description

Definition at line 892 of file ddi.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/hyp/ddi.c

8.55 NkDevOps Struct Reference

Data Fields

- unsigned long(∗ nk_grand_totalram_pages)(void)
- unsigned long(∗ nk_balloon_their_pages)(void)
- unsigned long(∗ nk_balloon_our_pages)(void)
- int(∗ nk_is_balloon_page)(unsigned long pfn)
- void(∗ nk_balloon_sysconf)(struct NkDevVlink ∗vlink)
- int(∗ nk_cons_hist_getchar)(nku64_f ∗ordnp)

  return one character from the history buffer.
- nku64_f(∗ nk_cons_hist_boot_ordinal)(void)

  return the ordinal number of the first character written in the history buffer after the last reboot.
- NkTracerFCallsRecorder(∗ nk_tracer_fcalls_recorder)(void)

  return the function that may be used as finction calls trace recorder
- void(∗ nk_xirq_mask_nosync)(NkXIrq xirq)

  Mask a given cross-interrupt. Does not wait for all the cross-interrupts handlers to complete.
- int(∗ nk_hv_version_get)(unsigned int ∗major, unsigned int ∗minor)

  return the hypervisor API version from the nk.version.api property.
- NkXIrqId(∗ nk_xirq_attach_named)(NkXIrq xirq, NkXIrqHandler hdl, void ∗cookie, char ∗name)

  Attach a handler to a given NanoKernel cross-interrupt. Must be called from base level. The handler is called with ONLY masked cross interrupt source.

8.55.1 Detailed Description

Definition at line 28 of file nkdevops.h.

8.55.2 Field Documentation

8.55.2.1 nk_grand_totalram_pages

```
unsigned long(∗ NkDevOps::nk_grand_totalram_pages) (void)
```

Returns the total size of this VM's RAM including all reserved memblocks

Definition at line 315 of file nkdevops.h.

8.55.2.2 nk_balloon_their_pages

```
unsigned long(* NkDevOps::nk_balloon_their_pages) (void)
```

Returns the number of balloon pages that initially belong to other guests

Definition at line 321 of file nkdevops.h.

8.55.2.3 nk_balloon_our_pages

```
unsigned long(* NkDevOps::nk_balloon_our_pages) (void)
```

Returns the number of balloon pages that initially belong to this guests

Definition at line 326 of file nkdevops.h.

8.55.2.4 nk_is_balloon_page

```
int(* NkDevOps::nk_is_balloon_page) (unsigned long pfn)
```

Check if the given pages belongs to a balloon memory region

Definition at line 330 of file nkdevops.h.

8.55.2.5 nk_balloon_sysconf

```
void(* NkDevOps::nk_balloon_sysconf) (struct NkDevVlink *vlink)
```

Called on each sysconf xirq

Definition at line 334 of file nkdevops.h.

8.55.2.6 nk_cons_hist_getchar

```
int(* NkDevOps::nk_cons_hist_getchar) (nku64_f *ordnp)
```

return one character from the history buffer.

Return the first available character stored in the history buffer under the ordinal number $*$*ordnp* or higher.

Parameters

in	$*$*ordnp*	the ordinal number of the requested character.
out	$*$*ordnp*	the ordinal number of the next character which may be already written in the buffer or not

Return values

-1	no characters available under the ordinal number $*$*ordnp* or higher.
$>$*0*	the requested character. The ordinal of the next character returned in $*$*ordnp*.

Definition at line 354 of file nkdevops.h.

8.55.2.7 nk_xirq_mask_nosync

```
void(* NkDevOps::nk_xirq_mask_nosync) (NkXIrq xirq)
```

Mask a given cross-interrupt. Does not wait for all the cross-interrupts handlers to complete.

In particular, can be called in the context of the cross-interrupt to be masked without any deadlock.

Definition at line 372 of file nkdevops.h.

8.55.2.8 nk_hv_version_get

```
int(* NkDevOps::nk_hv_version_get) (unsigned int *major, unsigned int *minor)
```

return the hypervisor API version from the nk.version.api property.

The property is in the form MAJOR.MINOR.VERSION.

Parameters

out	$*$*major*	the MAJOR component of the version.
out	$*$*minor*	the MINOR component of the version.

Return values

-1	the version wasn't returned due to an error.
0	the version has been parsed and returned through the output parameters.

Definition at line 386 of file nkdevops.h.

8.55.2.9 nk_xirq_attach_named

```
NkXIrqId(* NkDevOps::nk_xirq_attach_named) (NkXIrq xirq, NkXIrqHandler hdl, void *cookie, char
*name)
```

Attach a handler to a given NanoKernel cross-interrupt. Must be called from base level. The handler is called with ONLY masked cross interrupt source.

Return values

0	the handler can't be created, named or attached
>0	the cross-interrupt ID

Definition at line 395 of file nkdevops.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/nkdevops.h

8.56 NkDevRing Struct Reference

8.56.1 Detailed Description

Definition at line 444 of file nkdevops.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/nkdevops.h

8.57 nkregion_t Struct Reference

8.57.1 Detailed Description

Definition at line 26 of file nkregion.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/nkregion.h

8.58 NkStream Struct Reference

8.58.1 Detailed Description

Definition at line 62 of file vaudio.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio.c

8.59 NkVaudio Struct Reference

8.59.1 Detailed Description

This defines the vaudio types.

Definition at line 170 of file vaudio-fe.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-fe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio.c

8.60 NkVaudio.mixer Struct Reference

8.60.1 Detailed Description

Definition at line 174 of file vaudio-fe.c.

The documentation for this struct was generated from the following files:

8.61 NkVaudio.mixer.stats Struct Reference

8.61.1 Detailed Description

Definition at line 176 of file vaudio-fe.c.

The documentation for this struct was generated from the following files:

8.62 nvram_capsule_cmd Struct Reference

8.62.1 Detailed Description

Definition at line 68 of file vlx-bootloader-control.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-bootloader-control.c

8.63 nvram_msg Struct Reference

8.63.1 Detailed Description

Definition at line 87 of file vlx-bootloader-control.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-bootloader-control.c

8.64 nvram_reboot_cmd Struct Reference

8.64.1 Detailed Description

Definition at line 75 of file vlx-bootloader-control.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-bootloader-control.c

8.65 ofdyn_t Struct Reference

8.65.1 Detailed Description

Definition at line 73 of file vlx-ofdyn.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-ofdyn.c

8.66 pd_consumer Struct Reference

8.66.1 Detailed Description

Definition at line 44 of file vlx-pm-domain-test-consumer.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-test-consumer.c

8.67 pd_domain Struct Reference

8.67.1 Detailed Description

Definition at line 57 of file vlx-pm-domain-test-provider.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-test-provider.c

8.68 pd_provider Struct Reference

8.68.1 Detailed Description

Definition at line 49 of file vlx-pm-domain-test-provider.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-test-provider.c

8.69 peer_t Struct Reference

8.69.1 Detailed Description

Definition at line 87 of file vlx-clk-ctrl-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-fe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-fe.c

8.70 pfb_dev Struct Reference

8.70.1 Detailed Description

Definition at line 49 of file vlx-pfb.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pfb.c

8.71 pgtab_t Struct Reference

8.71.1 Detailed Description

Definition at line 94 of file pmem-user-arm.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pmem-user-arm.h

8.72 pip_rect_t Struct Reference

8.72.1 Detailed Description

Definition at line 20 of file vevdev-ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev-ioctl.h

8.73 pmd_update_t Struct Reference

8.73.1 Detailed Description

Definition at line 82 of file pmem-user-arm.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pmem-user-arm.h

8.74 pmd_update_t.__unnamed__ Union Reference

8.74.1 Detailed Description

Definition at line 85 of file pmem-user-arm.h.

The documentation for this union was generated from the following files:

8.75 pmem_mm_arch_t Struct Reference

8.75.1 Detailed Description

Definition at line 109 of file pmem-user-arm.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pmem-user-arm.h

8.76 pmem_region_arch_t Struct Reference

8.76.1 Detailed Description

Definition at line 100 of file pmem-user-arm.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pmem-user-arm.h

8.77 pmem_user_mngr_t Struct Reference

8.77.1 Detailed Description

Definition at line 108 of file pmem-user.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pmem-user.h

8.78 pmon_cpu_data_t Struct Reference

8.78.1 Detailed Description

Definition at line 97 of file perfmonitor.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/perfmonitor.c

8.79 pmon_percpu_t Struct Reference

8.79.1 Detailed Description

Definition at line 103 of file perfmonitor.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/perfmonitor.c

8.80 PmonCpuStats Struct Reference

8.80.1 Detailed Description

Definition at line 42 of file perfmon.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/perfmon.h

8.81 PmonSysInfo Struct Reference

8.81.1 Detailed Description

Definition at line 51 of file perfmon.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/perfmon.h

8.82 PmonTimerInfo Struct Reference

8.82.1 Detailed Description

Definition at line 37 of file perfmon.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/perfmon.h

8.83 policy_t Struct Reference

8.83.1 Detailed Description

Definition at line 209 of file vevdev-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-be.c

8.84 ppga_ex1_list_element_t Struct Reference

8.84.1 Detailed Description

Definition at line 51 of file ppga-example-1.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/ppga/ppga-example-1.c

8.85 ppga_ex1_list_t Struct Reference

8.85.1 Detailed Description

Definition at line 61 of file ppga-example-1.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/ppga/ppga-example-1.c

8.86 ppga_ex1_t Struct Reference

8.86.1 Detailed Description

Definition at line 77 of file ppga-example-1.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/ppga/ppga-example-1.c

8.87 ppga_t1_list_element_t Struct Reference

8.87.1 Detailed Description

Definition at line 38 of file ppga-test-1.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/ppga/ppga-test-1.c

8.88 ppga_t1_list_t Struct Reference

8.88.1 Detailed Description

Definition at line 46 of file ppga-test-1.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/ppga/ppga-test-1.c

8.89 ppga_t1_t Struct Reference

8.89.1 Detailed Description

Definition at line 59 of file ppga-test-1.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/ppga/ppga-test-1.c

8.90 provider_t Struct Reference

8.90.1 Detailed Description

Definition at line 74 of file vlx-clk-ctrl-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-fe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-test-provider.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain-fe.c

8.91 raudio_ops_t Struct Reference

8.91.1 Detailed Description

Definition at line 58 of file vaudio-raudio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-raudio.h

8.92 raudio_session_t Struct Reference

8.92.1 Detailed Description

Definition at line 53 of file vaudio-bridge.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-bridge.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-fake-native.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-loopback.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-user.c

8.93 raudio_session_t.dop Struct Reference

8.93.1 Detailed Description

Definition at line 178 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.94 raudio_session_t.stats Struct Reference

8.94.1 Detailed Description

Definition at line 189 of file vaudio-user.c.

The documentation for this struct was generated from the following files:

8.95 result_t Struct Reference

8.95.1 Detailed Description

Definition at line 58 of file vevdev-expr-test.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-expr-test.c

8.96 s2mpu_fault_information_v9 Struct Reference

8.96.1 Detailed Description

Definition at line 15 of file s2mpu-fault-info-v9.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/s2mpu-fault-info-v9.h

8.97 s2mpu_fault_notifier Struct Reference

8.97.1 Detailed Description

Definition at line 30 of file s2mpu-fault-info-priv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/s2mpu-fault-info-priv.h

8.98 s2mpu_fault_notifier_chain Struct Reference

8.98.1 Detailed Description

Definition at line 25 of file s2mpu-fault-info-priv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/s2mpu-fault-info-priv.h

8.99 s2mpu_sysconf_fault_handle Struct Reference

8.99.1 Detailed Description

Definition at line 59 of file s2mpu-fault-info-priv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/s2mpu-fault-info-priv.h

8.100 signal_t Struct Reference

8.100.1 Detailed Description

Definition at line 63 of file vlx-clk-ctrl-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-fe.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-ctrl-test-provider.c

8.101 signal_t.rate Union Reference

8.101.1 Detailed Description

Definition at line 75 of file vlx-clk-ctrl-test-provider.c.

The documentation for this union was generated from the following files:

8.102 signal_t.rate.fixed Struct Reference

8.102.1 Detailed Description

Definition at line 76 of file vlx-clk-ctrl-test-provider.c.

The documentation for this struct was generated from the following files:

8.103 signal_t.rate.fixed_factor Struct Reference

8.103.1 Detailed Description

Definition at line 80 of file vlx-clk-ctrl-test-provider.c.

The documentation for this struct was generated from the following files:

8.104 signal_t.rate.fractional_divider Struct Reference

8.104.1 Detailed Description

Definition at line 84 of file vlx-clk-ctrl-test-provider.c.

The documentation for this struct was generated from the following files:

8.105 sysctl_overlay Struct Reference

8.105.1 Detailed Description

Definition at line 63 of file vlx-sysctl-overlay.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-sysctl-overlay.c

8.106 sysctl_ovl_type_proc_handler Struct Reference

8.106.1 Detailed Description

Definition at line 434 of file vlx-sysctl-overlay.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-sysctl-overlay.c

8.107 timing_cpu_t Struct Reference

8.107.1 Detailed Description

Definition at line 67 of file vlx-timing.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-timing.c

8.108 timing_reg_t Struct Reference

8.108.1 Detailed Description

Definition at line 56 of file vlx-timing.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-timing.c

8.109 ued_t Struct Reference

8.109.1 Detailed Description

Definition at line 67 of file uevent-dump.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/uevent-dump.c

8.110 update_t Struct Reference

8.110.1 Detailed Description

Definition at line 58 of file vlx-ofdyn.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-ofdyn.c

8.111 value_t Union Reference

8.111.1 Detailed Description

Definition at line 51 of file vevdev-expr-test.c.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-expr-test.c

8.112 vaudio_be_t Struct Reference

8.112.1 Detailed Description

Definition at line 172 of file vaudio-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-be.c

8.113 vaudio_command_t Struct Reference

8.113.1 Detailed Description

Definition at line 135 of file vaudio-user.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-user.c

8.114 vaudio_command_t.params Union Reference

8.114.1 Detailed Description

Definition at line 137 of file vaudio-user.c.

The documentation for this union was generated from the following files:

8.115 vaudio_ctrl_params_t Struct Reference

8.115.1 Detailed Description

Definition at line 120 of file vaudio-user.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-user.c

8.116 vaudio_ctrl_params_t.params Union Reference

8.116.1 Detailed Description

Definition at line 122 of file vaudio-user.c.

The documentation for this union was generated from the following files:

8.117 vaudio_fe_t Struct Reference

8.117.1 Detailed Description

Definition at line 216 of file vaudio-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-be.c

8.118 vaudio_fe_t.stats Struct Reference

8.118.1 Detailed Description

Definition at line 235 of file vaudio-be.c.

The documentation for this struct was generated from the following files:

8.119 vaudio_result_t Struct Reference

8.119.1 Detailed Description

Definition at line 144 of file vaudio-user.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-user.c

8.120 vaudio_start_params_t Struct Reference

8.120.1 Detailed Description

Definition at line 110 of file vaudio-user.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-user.c

8.121 vaudio_stream_params_t Struct Reference

8.121.1 Detailed Description

Definition at line 96 of file vaudio-user.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-user.c

8.122 vaudio_stream_t Struct Reference

8.122.1 Detailed Description

Definition at line 147 of file vaudio-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio-fe.c

8.123 vaudio_stream_t.stats Struct Reference

8.123.1 Detailed Description

Definition at line 141 of file vaudio-fe.c.

The documentation for this struct was generated from the following files:

8.124 vaudio_stream_t.stats Struct Reference

8.124.1 Detailed Description

Definition at line 141 of file vaudio-fe.c.

The documentation for this struct was generated from the following files:

8.125 vbd2_get_geo Struct Reference

8.125.1 Detailed Description

Definition at line 161 of file vbd2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vbd2_common.h

8.126 vbd2_msg Union Reference

8.126.1 Detailed Description

Definition at line 193 of file vbd2_common.h.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vbd2_common.h

8.127 vbd2_probe Struct Reference

8.127.1 Detailed Description

Definition at line 176 of file vbd2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vbd2_common.h

8.128 vbd2_probe_link Struct Reference

8.128.1 Detailed Description

Definition at line 186 of file vbd2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vbd2_common.h

8.129 vbd2_req_header Struct Reference

8.129.1 Detailed Description

Definition at line 95 of file vbd2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vbd2_common.h

8.130 vbd_be Struct Reference

8.130.1 Detailed Description

Definition at line 430 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-be.c

8.131 vbd_be.blkio_thread Struct Reference

8.131.1 Detailed Description

Definition at line 436 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.132 vbd_be.blkio_thread.stats Struct Reference

8.132.1 Detailed Description

Definition at line 442 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.133 vbd_be.event_thread Struct Reference

8.133.1 Detailed Description

Definition at line 449 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.134 vbd_be.event_thread.stats Struct Reference

8.134.1 Detailed Description

Definition at line 456 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.135 vbd_disk Struct Reference

8.135.1 Detailed Description

Definition at line 170 of file vlx-vbd2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-fe.c

8.136 vbd_disk.stats Struct Reference

8.136.1 Detailed Description

Definition at line 187 of file vlx-vbd2-fe.c.

The documentation for this struct was generated from the following files:

8.137 vbd_disk_match Struct Reference

8.137.1 Detailed Description

Definition at line 866 of file vlx-vbd2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-fe.c

8.138 vbd_fe Struct Reference

8.138.1 Detailed Description

Definition at line 290 of file vlx-vbd2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-fe.c

8.139 vbd_link Struct Reference

8.139.1 Detailed Description

Definition at line 291 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-fe.c

8.140 vbd_link.done_list Struct Reference

8.140.1 Detailed Description

Definition at line 306 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.141 vbd_link.stats Struct Reference

8.141.1 Detailed Description

Definition at line 316 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.142 vbd_link.stats.dma_ls Struct Reference

8.142.1 Detailed Description

Definition at line 334 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.143 vbd_link.stats.dma_vm Struct Reference

8.143.1 Detailed Description

Definition at line 338 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.144 vbd_link.stats.ls_map Struct Reference

8.144.1 Detailed Description

Definition at line 318 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.145 vbd_link.stats.nk_mem_map Struct Reference

8.145.1 Detailed Description

Definition at line 329 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.146 vbd_major Struct Reference

8.146.1 Detailed Description

Definition at line 162 of file vlx-vbd2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-fe.c

8.147 vbd_options Struct Reference

8.147.1 Detailed Description

Definition at line 36 of file vbd2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vbd2_common.h

8.148 vbd_options.granting Struct Reference

8.148.1 Detailed Description

Definition at line 42 of file vbd2_common.h.

The documentation for this struct was generated from the following files:

8.149 vbd_pending_req Struct Reference

8.149.1 Detailed Description

Definition at line 507 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-be.c

8.150 vbd_pending_seg Struct Reference

8.150.1 Detailed Description

Definition at line 495 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-be.c

8.151 vbd_prop_vdisk Struct Reference

8.151.1 Detailed Description

Definition at line 250 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-be.c

8.152 vbd_req Struct Reference

8.152.1 Detailed Description

Definition at line 204 of file vlx-vbd2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-fe.c

8.153 vbd_rgntab Struct Reference

8.153.1 Detailed Description

Definition at line 37 of file vlx-vbd2-common.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-common.c

8.154 vbd_rgntab_outer Struct Reference

8.154.1 Detailed Description

Definition at line 45 of file vlx-vbd2-common.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-common.c

8.155 vbd_type Struct Reference

8.155.1 Detailed Description

Definition at line 153 of file vlx-vbd2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-fe.c

8.156 vbd_vdisk Struct Reference

8.156.1 Detailed Description

Definition at line 390 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vbd2-be.c

8.157 vbd_vdisk.stats Struct Reference

8.157.1 Detailed Description

Definition at line 403 of file vlx-vbd2-be.c.

The documentation for this struct was generated from the following files:

8.158 vbufq_be_context_t Struct Reference

Represent a user process.

```
#include <vbufq-be.h>
```

Data Fields

- char * name
- vbufq_link_t * vbufq_link
- struct file * filp
- void * signature

8.158.1 Detailed Description

Represent a user process.

Definition at line 229 of file vbufq-be.h.

8.158.2 Field Documentation

8.158.2.1 name

```
char* vbufq_be_context_t::name
```

name of the related process

Definition at line 230 of file vbufq-be.h.

8.158.2.2 vbufq_link

`vbufq_link_t* vbufq_be_context_t::vbufq_link`

vbufq_link to which the context is related to

Definition at line 232 of file vbufq-be.h.

8.158.2.3 filp

`struct file* vbufq_be_context_t::filp`

file descriptor for bufq device interface

Definition at line 233 of file vbufq-be.h.

8.158.2.4 signature

`void* vbufq_be_context_t::signature`

permits to check the consistency of a ctx

Definition at line 237 of file vbufq-be.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-be.h

8.159 vbufq_be_t Struct Reference

Global data in the driver. A global instance is present the driver.

`#include <vbufq-be.h>`

Data Fields

- struct semaphore thread_sem
- _Bool is_thread_aborted
- _Bool is_sysconf
- vlx_thread_t vmq_thread_desc
- vmq_links_t ∗ links

8.159.1 Detailed Description

Global data in the driver. A global instance is present the driver.

Definition at line 163 of file vbufq-be.h.

8.159.2 Field Documentation

8.159.2.1 thread_sem

`struct semaphore vbufq_be_t::thread_sem`

semaphore used to wakeup the VMQ housekeeping thread

Definition at line 164 of file vbufq-be.h.

8.159.2.2 is_thread_aborted

`_Bool vbufq_be_t::is_thread_aborted`

notify is the VMQ housekeeping thread has been waked up in order to abort.

Definition at line 166 of file vbufq-be.h.

8.159.2.3 is_sysconf

`_Bool vbufq_be_t::is_sysconf`

notify is the VMQ housekeeping thread has been waked up for a sysconf event

Definition at line 167 of file vbufq-be.h.

8.159.2.4 vmq_thread_desc

`vlx_thread_t vbufq_be_t::vmq_thread_desc`

thread descriptor for the VMQ housekeeping thread

Definition at line 169 of file vbufq-be.h.

8.159.2.5 links

```
vmq_links_t* vbufq_be_t::links
```

list of all VMQ links used

Definition at line 170 of file vbufq-be.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-be.h

8.160 vbufq_device_t Struct Reference

8.160.1 Detailed Description

Definition at line 111 of file vbufq-fe.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-fe.h

8.161 vbufq_fe_context_t Struct Reference

Represent a user process.

```
#include <vbufq-fe.h>
```

Data Fields

- vbufq_link_t * vbufq_link
- char * name
- struct list_head node
- void * cookie
- void * signature
- unsigned int req_cmpt
- _Bool is_valid

8.161.1 Detailed Description

Represent a user process.

Definition at line 169 of file vbufq-fe.h.

8.161.2 Field Documentation

8.161.2.1 vbufq_link

`vbufq_link_t* vbufq_fe_context_t::vbufq_link`

vbufq_link to which the context is related to

Definition at line 171 of file vbufq-fe.h.

8.161.2.2 name

`char* vbufq_fe_context_t::name`

name of the related process

Definition at line 172 of file vbufq-fe.h.

8.161.2.3 node

`struct list_head vbufq_fe_context_t::node`

permits to keep track of the context in the ctx_list of a vbufq_link_t structure

Definition at line 173 of file vbufq-fe.h.

8.161.2.4 cookie

`void* vbufq_fe_context_t::cookie`

RW by be, permits to get the be ctx associated with a fe ctx

Definition at line 175 of file vbufq-fe.h.

8.161.2.5 signature

`void* vbufq_fe_context_t::signature`

permits to check the consistency of a ctx

Definition at line 176 of file vbufq-fe.h.

8.161.2.6 req_cmpt

```
unsigned int vbufq_fe_context_t::req_cmpt
```

number of requests issued by the process, for debugging log purpose

Definition at line 178 of file vbufq-fe.h.

8.161.2.7 is_valid

```
_Bool vbufq_fe_context_t::is_valid
```

false if a backend disconnection occurred on its vbufq_link, true otherwise

Definition at line 179 of file vbufq-fe.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-fe.h

8.162 vbufq_fe_t Struct Reference

Global data in the driver. A global instance is present the driver.

```
#include <vbufq-fe.h>
```

Data Fields

- struct semaphore thread_sem
- _Bool is_thread_aborted
- _Bool is_sysconf
- vlx_thread_t thread_desc
- vmq_links_t * links
- vbufq_device_t info_devices

8.162.1 Detailed Description

Global data in the driver. A global instance is present the driver.

Definition at line 122 of file vbufq-fe.h.

8.162.2 Field Documentation

8.162.2.1 thread_sem

`struct semaphore vbufq_fe_t::thread_sem`

semaphore used to wakeup the VMQ housekeeping thread

Definition at line 123 of file vbufq-fe.h.

8.162.2.2 is_thread_aborted

`_Bool vbufq_fe_t::is_thread_aborted`

notify is the VMQ housekeeping thread has been waked up in order to abort.

Definition at line 125 of file vbufq-fe.h.

8.162.2.3 is_sysconf

`_Bool vbufq_fe_t::is_sysconf`

notify is the VMQ housekeeping thread has been waked up for a sysconf event

Definition at line 126 of file vbufq-fe.h.

8.162.2.4 thread_desc

`vlx_thread_t vbufq_fe_t::thread_desc`

thread descriptor for the VMQ housekeeping thread

Definition at line 128 of file vbufq-fe.h.

8.162.2.5 links

`vmq_links_t* vbufq_fe_t::links`

list of all VMQ links used

Definition at line 129 of file vbufq-fe.h.

8.162.2.6 info_devices

```
vbufq_device_t vbufq_fe_t::info_devices
```

contains info related to the class device created for the fe.

Definition at line 131 of file vbufq-fe.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-fe.h

8.163 vbufq_link_t Struct Reference

Represent a connection between a frontend and a backend.

```
#include <vbufq-be.h>
```

Data Fields

- vmq_link_t ∗ link
- _Bool is_thread_aborted
- struct mutex mutex
- struct radix_tree_root ctx_tree
- _Bool is_up
- struct workqueue_struct ∗ wq
- vbufq_msg_box_generator_t gen
- char ∗ device_name
- char ∗ real_device_name
- vmq_xx_config_t xx_config
- struct work_struct feeder_work

8.163.1 Detailed Description

Represent a connection between a frontend and a backend.

Definition at line 185 of file vbufq-be.h.

8.163.2 Field Documentation

8.163.2.1 link

```
vmq_link_t* vbufq_link_t::link
```

VMQ link representing the connection

Definition at line 186 of file vbufq-be.h.

8.163.2.2 is_thread_aborted

```
_Bool vbufq_link_t::is_thread_aborted
```

notify if the feeding thread has been waked up in order to abort.

Definition at line 187 of file vbufq-be.h.

8.163.2.3 mutex

```
struct mutex vbufq_link_t::mutex
```

mutex permitting to control access to the ctx_tree.

Definition at line 188 of file vbufq-be.h.

8.163.2.4 ctx_tree

```
struct radix_tree_root vbufq_link_t::ctx_tree
```

contains all contexts related to this connection

Definition at line 190 of file vbufq-be.h.

8.163.2.5 is_up

```
_Bool vbufq_link_t::is_up
```

notify if the connection is on, i.e fe,be have detected each other.

Definition at line 192 of file vbufq-be.h.

8.163.2.6 wq

```
struct workqueue_struct* vbufq_link_t::wq
```

workqueue handling the forwarding of requests/responses to/from between bufq and fe.

Definition at line 197 of file vbufq-be.h.

8.163.2.7 gen

`vbufq_msg_box_generator_t vbufq_link_t::gen`

message box generator used for the work pool management.

Definition at line 201 of file vbufq-be.h.

8.163.2.8 device_name

`char* vbufq_link_t::device_name`

name of the char device created to reach the fe related to this connection.

Definition at line 202 of file vbufq-be.h.

8.163.2.9 real_device_name

`char* vbufq_link_t::real_device_name`

name of the backend char device related to this connection.

Definition at line 203 of file vbufq-be.h.

8.163.2.10 xx_config

`vmq_xx_config_t vbufq_link_t::xx_config`

contains VMQ configuration, dynamically get from the device tree, for rx configuration.

Definition at line 205 of file vbufq-be.h.

8.163.2.11 feeder_work

`struct work_struct vbufq_link_t::feeder_work`

work object dedicated for the feeder of this vbufq_link

Definition at line 210 of file vbufq-be.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-be.h

8.164 vbufq_link_t Struct Reference

Represent a connection between a frontend and a backend.

```
#include <vbufq-fe.h>
```

Data Fields

- vmq_link_t * link
- vipc_ctx_t vipc_ctx
- _Bool is_up
- spinlock_t slock
- struct list_head ctx_list
- unsigned int dev_minor
- char * device_name
- char * real_device_name
- vmq_xx_config_t xx_config
- _Bool requests_are_allowed
- wait_queue_head_t wait_queue

8.164.1 Detailed Description

Represent a connection between a frontend and a backend.

Definition at line 143 of file vbufq-fe.h.

8.164.2 Field Documentation

8.164.2.1 link

```
vmq_link_t* vbufq_link_t::link
```

VMQ link representing the connection

Definition at line 144 of file vbufq-fe.h.

8.164.2.2 vipc_ctx

```
vipc_ctx_t vbufq_link_t::vipc_ctx
```

used by the vipc library, permitting to issue synchronous requests with VMQ

Definition at line 146 of file vbufq-fe.h.

8.164.2.3 is_up

```
_Bool vbufq_link_t::is_up
```

notify if the connection is on, i.e fe,be have detected each other.

Definition at line 147 of file vbufq-fe.h.

8.164.2.4 slock

```
spinlock_t vbufq_link_t::slock
```

spinlock permitting to control access to the ctx_list

Definition at line 149 of file vbufq-fe.h.

8.164.2.5 ctx_list

```
struct list_head vbufq_link_t::ctx_list
```

list of all contexts related to this connection

Definition at line 150 of file vbufq-fe.h.

8.164.2.6 dev_minor

```
unsigned int vbufq_link_t::dev_minor
```

minor of the char device created to reach the fe related to this connection

Definition at line 153 of file vbufq-fe.h.

8.164.2.7 device_name

```
char* vbufq_link_t::device_name
```

name of the char device created to reach the fe related to this connection

Definition at line 154 of file vbufq-fe.h.

8.164.2.8 real_device_name

`char* vbufq_link_t::real_device_name`

name of the backend char device related to this connection.

Definition at line 155 of file vbufq-fe.h.

8.164.2.9 xx_config

`vmq_xx_config_t vbufq_link_t::xx_config`

contains VMQ configuration, dynamically get from the device tree, for tx configuration

Definition at line 157 of file vbufq-fe.h.

8.164.2.10 requests_are_allowed

`_Bool vbufq_link_t::requests_are_allowed`

notify if new requests coming from user process are allowed or not.

Definition at line 158 of file vbufq-fe.h.

8.164.2.11 wait_queue

`wait_queue_head_t vbufq_link_t::wait_queue`

used for poll, waiting for be connection

Definition at line 160 of file vbufq-fe.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-fe.h

8.165 vbufq_msg_box_generator_t Struct Reference

describe a message box generator

`#include <vbufq-be.h>`

Data Fields

- vbufq_stack_t ready2use
- unsigned int nb_works

8.165.1 Detailed Description

describe a message box generator

Definition at line 154 of file vbufq-be.h.

8.165.2 Field Documentation

8.165.2.1 ready2use

```
vbufq_stack_t vbufq_msg_box_generator_t::ready2use
```

stack containing msg_box ready to use, released by producers

Definition at line 156 of file vbufq-be.h.

8.165.2.2 nb_works

```
unsigned int vbufq_msg_box_generator_t::nb_works
```

number of mesg box created

Definition at line 157 of file vbufq-be.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-be.h

8.166 vbufq_msg_box_t Struct Reference

container created in order to embed a message, used by thread of work queues.

```
#include <vbufq-be.h>
```

Data Fields

- vbufq_msg_t ∗ msg
- struct work_struct work
- vbufq_link_t ∗ vbufq_link

8.166.1 Detailed Description

container created in order to embed a message, used by thread of work queues.

Thus, msg is reachable from work callbacks, using the macro container_of

Definition at line 122 of file vbufq-be.h.

8.166.2 Field Documentation

8.166.2.1 msg

`vbufq_msg_t* vbufq_msg_box_t::msg`

message coming from the fe

Definition at line 123 of file vbufq-be.h.

8.166.2.2 work

`struct work_struct vbufq_msg_box_t::work`

work object that will be provided to the workqueue

Definition at line 128 of file vbufq-be.h.

8.166.2.3 vbufq_link

`vbufq_link_t* vbufq_msg_box_t::vbufq_link`

permits to notify to workqueue callbacks from which VM the message is coming from

Definition at line 132 of file vbufq-be.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-be.h

8.167 vbufq_msg_header Struct Reference

8.167.1 Detailed Description

Definition at line 52 of file vbufq_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq_common.h

8.168 vbufq_msg_status Struct Reference

8.168.1 Detailed Description

Definition at line 64 of file vbufq_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq_common.h

8.169 vbufq_msg_t Struct Reference

8.169.1 Detailed Description

Definition at line 77 of file vbufq_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq_common.h

8.170 vbufq_payload_t Union Reference

8.170.1 Detailed Description

Definition at line 69 of file vbufq_common.h.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq_common.h

8.171 vbufq_stack_element_t Struct Reference

an element of a vbufq_stack_t.

```
#include <vbufq-be.h>
```

Data Fields

- struct vbufq_stack_element_t * next
- vbufq_msg_box_t msg_box

8.171.1 Detailed Description

an element of a vbufq_stack_t.

Definition at line 138 of file vbufq-be.h.

8.171.2 Field Documentation

8.171.2.1 next

```
struct vbufq_stack_element_t* vbufq_stack_element_t::next
```

next element of the stack

Definition at line 139 of file vbufq-be.h.

8.171.2.2 msg_box

```
vbufq_msg_box_t vbufq_stack_element_t::msg_box
```

were useful data are embedded

Definition at line 140 of file vbufq-be.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-be.h

8.172 vbufq_stack_t Struct Reference

structure used for a stack implementation

```
#include <vbufq-be.h>
```

Data Fields

- vbufq_stack_element_t ∗ first

8.172.1 Detailed Description

structure used for a stack implementation

Definition at line 146 of file vbufq-be.h.

8.172.2 Field Documentation

8.172.2.1 first

```
vbufq_stack_element_t* vbufq_stack_t::first
```

point to the first element of the stack

Definition at line 148 of file vbufq-be.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vbufq-be.h

8.173 vclk_docile_provider_t Struct Reference

8.173.1 Detailed Description

Definition at line 49 of file vlx-clk-docile.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-docile.c

8.174 vclk_docile_t Struct Reference

8.174.1 Detailed Description

Definition at line 56 of file vlx-clk-docile.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-clk-docile.c

8.175 vcpu_info_t Struct Reference

8.175.1 Detailed Description

Definition at line 42 of file pm-hmp.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/pm-hmp.c

8.176 VdelayDev Struct Reference

8.176.1 Detailed Description

Definition at line 43 of file vdelay.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdelay.c

8.177 Vdev Struct Reference

8.177.1 Detailed Description

Definition at line 118 of file vevdev-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vevdev-fe.c

8.178 vdmaheap_buffer Struct Reference

(a.k.a. vbuf) represents the original dmabuf (i.e. the exported dmabuf).

Data Fields

- struct list_head vbuf_node
- struct kref kref
- struct vdmaheap_completion completion
- int errno
- struct mutex cred_list_lock
- vdmaheap_buffer_gid_t gid
- struct dma_buf ∗ dmabuf
- struct sg_table ∗ sg_table
- struct vdmaheap_sg ∗ sg
- unsigned int sg_size
- struct vdmaheap_client ∗ client
- struct list_head cred_list
- nku32_f granted [NK_OS_LIMIT]
- unsigned long heap_flags
- unsigned long grace_max

8.178.1 Detailed Description

(a.k.a. vbuf) represents the original dmabuf (i.e. the exported dmabuf).

Definition at line 57 of file vdmaheap_export.c.

8.178.2 Field Documentation

8.178.2.1 vbuf_node

`struct list_head vdmaheap_buffer::vbuf_node`

node for the vdmaheap_device list

Definition at line 59 of file vdmaheap_export.c.

8.178.2.2 kref

`struct kref vdmaheap_buffer::kref`

reference counter held by vdmaheap_cred and the idr_tree

Definition at line 62 of file vdmaheap_export.c.

8.178.2.3 completion

`struct vdmaheap_completion vdmaheap_buffer::completion`

concurrent exporters are blocked on this until the vbuf creation is completed

Definition at line 64 of file vdmaheap_export.c.

8.178.2.4 errno

`int vdmaheap_buffer::errno`

indicates the completion status to a concurrent exporter

Definition at line 66 of file vdmaheap_export.c.

8.178.2.5 cred_list_lock

`struct mutex vdmaheap_buffer::cred_list_lock`

protects against concurrent accesses to the cred list

Definition at line 68 of file vdmaheap_export.c.

8.178.2.6 gid

`vdmaheap_buffer_gid_t vdmaheap_buffer::gid`

Global buffer IDentifier (VM id + local buffer id)

Definition at line 70 of file vdmaheap_export.c.

8.178.2.7 dmabuf

`struct dma_buf* vdmaheap_buffer::dmabuf`

points to the original dma_buf object

Definition at line 72 of file vdmaheap_export.c.

8.178.2.8 sg_table

`struct sg_table* vdmaheap_buffer::sg_table`

the kernel object for the dmabuf memory layout

Definition at line 75 of file vdmaheap_export.c.

8.178.2.9 sg

`struct vdmaheap_sg* vdmaheap_buffer::sg`

the vdmaheap object for the dmabuf memory layout serialized from the sg_table

Definition at line 77 of file vdmaheap_export.c.

8.178.2.10 sg_size

```
unsigned int vdmaheap_buffer::sg_size
```

size of the sg_table

Definition at line 79 of file vdmaheap_export.c.

8.178.2.11 client

```
struct vdmaheap_client* vdmaheap_buffer::client
```

exporter client (the first one)

Definition at line 81 of file vdmaheap_export.c.

8.178.2.12 cred_list

```
struct list_head vdmaheap_buffer::cred_list
```

list of export credentials associated with that buffer

Definition at line 83 of file vdmaheap_export.c.

8.178.2.13 granted

```
nku32_f vdmaheap_buffer::granted[NK_OS_LIMIT]
```

per-vm grant access flags

Definition at line 85 of file vdmaheap_export.c.

8.178.2.14 heap_flags

```
unsigned long vdmaheap_buffer::heap_flags
```

flags of the heap the dmabuf was allocated in

Definition at line 87 of file vdmaheap_export.c.

8.178.2.15 grace_max

`unsigned long vdmaheap_buffer::grace_max`

maximal grace delay allowed by the dmabuf exporters clients

Definition at line 89 of file vdmaheap_export.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_export.c

8.179 vdmaheap_cmd Struct Reference

8.179.1 Detailed Description

Definition at line 63 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.180 vdmaheap_cmd_res_release Struct Reference

8.180.1 Detailed Description

Definition at line 134 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.181 vdmaheap_completion Struct Reference

8.181.1 Detailed Description

Definition at line 123 of file vdmaheap_dev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_dev.h

8.182 vdmaheap_device Struct Reference

vdmaheap_device data structure.

`#include <vdmaheap_dev.h>`

Data Fields

- atomic_t link_state
- wait_queue_head_t link_wq
- struct list_head client_list
- struct list_head vbuf_list
- struct list_head lru_list
- struct list_head be_list
- struct list_head fe_list
- struct list_head dmabuf_ops_list
- struct idr vbuf_idr
- struct radix_tree_root vbuf_tree
- struct radix_tree_root vimp_tree
- struct mutex lock
- struct mutex lru_list_lock
- struct mutex vimp_tree_lock
- struct mutex client_list_lock
- struct mutex vbuf_list_lock
- struct mutex vbuf_tree_lock
- struct mutex vbuf_idr_lock
- struct mutex dmabuf_ops_lock
- struct vdmaheap_heap * vdmaheap_heap
- unsigned long lru_size_max
- unsigned long lru_size
- unsigned int vrpcs_minsize
- atomic_t vbuf_create_count
- atomic_t vbuf_release_count
- atomic_t client_create_count
- atomic_t client_release_count
- atomic_t vimp_create_count
- atomic_t vimp_release_count
- atomic_t dmabuf_create_count
- atomic_t dmabuf_release_count
- atomic_t cred_create_count
- atomic_t cred_release_count
- struct {
  atomic_t **op_client_create**
  atomic_t **op_client_create_nok**
  atomic_t **op_client_create_exited**
  atomic_t **op_client_destroy**
  atomic_t **op_client_destroy_exited**
  atomic_t **op_unexport**
  atomic_t **op_unexport_nok**
  atomic_t **op_unexport_exited**
  atomic_t **op_export**
  atomic_t **op_export_nok**
  atomic_t **op_export_exited**
  atomic_t **op_import**

atomic_t **op_import_nok**
atomic_t **op_import_exited**
atomic_t **op_release**
atomic_t **op_release_nok**
atomic_t **op_release_exited**
atomic_t **op_peer_release**
atomic_t **op_peer_release_nok**
atomic_t **op_peer_release_exited**
atomic_t **op_peer_import**
atomic_t **op_peer_import_nok**
atomic_t **op_peer_import_exited**
atomic_t **vbuf_reuse**
atomic_t **vbuf_create**
atomic_t **dmabuf_import**
atomic_t **dmabuf_import_reused**
atomic_t **dmabuf_import_created**
atomic_t **dmabuf_import_cache**
atomic_t **dmabuf_import_nok**
atomic_t **dmabuf_import_sent**
atomic_t **dmabuf_release_alive**
atomic_t **dmabuf_release_cacheable**
atomic_t **dmabuf_release_cached**
atomic_t **dmabuf_release_evicted**
atomic_t **dma_buf_attach**
atomic_t **dma_buf_detach**
atomic_t **sg_table_vmalloc**
atomic_t **sg_table_vfree**
atomic_t **vdmaheap_sg_alloc**
atomic_t **vdmaheap_sg_free**
struct avg_stats **buf_per_call**
struct avg_stats **buf_per_rsp**
struct avg_stats **reass_rsp_size**
struct avg_stats **rsp_size**
struct avg_stats **frag_per_rsp**
} stats

8.182.1 Detailed Description

vdmaheap_device data structure.

the top object in the vdmaheap hierarchy.

Definition at line 165 of file vdmaheap_dev.h.

8.182.2 Field Documentation

8.182.2.1 link_state

```
atomic_t vdmaheap_device::link_state
```

indicates the state of all vdmaheap links with the peer entities in the form of a bitmap. The bit i indicates the state of the vdmaheap link with the peer domain i:

- 1: the link is up.
- 0: the link is down. When a link is reported in the down state, all dmabufs imported from the corresponding domain must be closed by their owners.

Definition at line 179 of file vdmaheap_dev.h.

8.182.2.2 link_wq

```
wait_queue_head_t vdmaheap_device::link_wq
```

kernel wait_queue object associated with the link_state

Definition at line 181 of file vdmaheap_dev.h.

8.182.2.3 client_list

```
struct list_head vdmaheap_device::client_list
```

references all the vdmaheap_device vdmaheap_client objects

Definition at line 185 of file vdmaheap_dev.h.

8.182.2.4 vbuf_list

```
struct list_head vdmaheap_device::vbuf_list
```

references all the vdmaheap_device vdmaheap_buffer objects

Definition at line 187 of file vdmaheap_dev.h.

8.182.2.5 lru_list

```
struct list_head vdmaheap_device::lru_list
```

references all the vdmaheap_device vdmaheap_import objects (cache)

Definition at line 189 of file vdmaheap_dev.h.

8.182.2.6 be_list

```
struct list_head vdmaheap_device::be_list
```

references all the vdmaheap_device vdmaheap_be objects

Definition at line 191 of file vdmaheap_dev.h.

8.182.2.7 fe_list

```
struct list_head vdmaheap_device::fe_list
```

references all the vdmaheap_device vdmaheap_fe objects

Definition at line 193 of file vdmaheap_dev.h.

8.182.2.8 dmabuf_ops_list

```
struct list_head vdmaheap_device::dmabuf_ops_list
```

references all the vdmaheap_device templates for the caught dmabufs

Definition at line 195 of file vdmaheap_dev.h.

8.182.2.9 vbuf_idr

```
struct idr vdmaheap_device::vbuf_idr
```

references by their GID the vdmaheap_device vdmaheap_buffer objects

Definition at line 197 of file vdmaheap_dev.h.

8.182.2.10 vbuf_tree

`struct radix_tree_root vdmaheap_device::vbuf_tree`

references by their dmabuf address the vdmaheap_device vdmaheap_buffer objects

Definition at line 199 of file vdmaheap_dev.h.

8.182.2.11 vimp_tree

`struct radix_tree_root vdmaheap_device::vimp_tree`

references by their GID the vdmaheap_device vdmaheap_import objects

Definition at line 201 of file vdmaheap_dev.h.

8.182.2.12 lock

`struct mutex vdmaheap_device::lock`

protects this data structure against concurrent accesses

Definition at line 205 of file vdmaheap_dev.h.

8.182.2.13 lru_list_lock

`struct mutex vdmaheap_device::lru_list_lock`

protects the lru list against concurrent accesses

Definition at line 207 of file vdmaheap_dev.h.

8.182.2.14 vimp_tree_lock

`struct mutex vdmaheap_device::vimp_tree_lock`

protects vimp_tree against concurrent accesses

Definition at line 209 of file vdmaheap_dev.h.

8.182.2.15 client_list_lock

```
struct mutex vdmaheap_device::client_list_lock
```

protects client_list against concurrent accesses

Definition at line 211 of file vdmaheap_dev.h.

8.182.2.16 vbuf_list_lock

```
struct mutex vdmaheap_device::vbuf_list_lock
```

protects vbuf_list against concurrent accesses

Definition at line 213 of file vdmaheap_dev.h.

8.182.2.17 vbuf_tree_lock

```
struct mutex vdmaheap_device::vbuf_tree_lock
```

protects vbuf_tree against concurrent accesses

Definition at line 215 of file vdmaheap_dev.h.

8.182.2.18 vbuf_idr_lock

```
struct mutex vdmaheap_device::vbuf_idr_lock
```

protects vbuf_idr against concurrent accesses

Definition at line 217 of file vdmaheap_dev.h.

8.182.2.19 dmabuf_ops_lock

```
struct mutex vdmaheap_device::dmabuf_ops_lock
```

protects dmabuf_ops_list against concurrent accesses

Definition at line 219 of file vdmaheap_dev.h.

8.182.2.20 vdmaheap_heap

`struct vdmaheap_heap* vdmaheap_device::vdmaheap_heap`

pseudo heap featuring ops for imported remote buffers

Definition at line 222 of file vdmaheap_dev.h.

8.182.2.21 lru_size_max

`unsigned long vdmaheap_device::lru_size_max`

maximal cumulated size of the dmabufs "in-flight" in the cache

Definition at line 225 of file vdmaheap_dev.h.

8.182.2.22 lru_size

`unsigned long vdmaheap_device::lru_size`

current cumulated size of the dmabufs "in-flight" in the cache

Definition at line 227 of file vdmaheap_dev.h.

8.182.2.23 vrpcs_minsize

`unsigned int vdmaheap_device::vrpcs_minsize`

minimal size for the pmem

Definition at line 231 of file vdmaheap_dev.h.

8.182.2.24 vbuf_create_count

`atomic_t vdmaheap_device::vbuf_create_count`

number of created vdmaheap_buffer objects

Definition at line 235 of file vdmaheap_dev.h.

8.182.2.25 vbuf_release_count

```
atomic_t vdmaheap_device::vbuf_release_count
```

number of released vdmaheap_buffer objects

Definition at line 237 of file vdmaheap_dev.h.

8.182.2.26 client_create_count

```
atomic_t vdmaheap_device::client_create_count
```

number of created vdmaheap_client objects

Definition at line 239 of file vdmaheap_dev.h.

8.182.2.27 client_release_count

```
atomic_t vdmaheap_device::client_release_count
```

number of released vdmaheap_client objects

Definition at line 241 of file vdmaheap_dev.h.

8.182.2.28 vimp_create_count

```
atomic_t vdmaheap_device::vimp_create_count
```

number of created vdmaheap_import objects

Definition at line 243 of file vdmaheap_dev.h.

8.182.2.29 vimp_release_count

```
atomic_t vdmaheap_device::vimp_release_count
```

number of released vdmaheap_import objects

Definition at line 245 of file vdmaheap_dev.h.

8.182.2.30 dmabuf_create_count

```
atomic_t vdmaheap_device::dmabuf_create_count
```

number of created virtual dmabufs objects

Definition at line 247 of file vdmaheap_dev.h.

8.182.2.31 dmabuf_release_count

```
atomic_t vdmaheap_device::dmabuf_release_count
```

number of released virtual dmabufs objects

Definition at line 249 of file vdmaheap_dev.h.

8.182.2.32 cred_create_count

```
atomic_t vdmaheap_device::cred_create_count
```

number of created vdmaheap_cred objects

Definition at line 251 of file vdmaheap_dev.h.

8.182.2.33 cred_release_count

```
atomic_t vdmaheap_device::cred_release_count
```

number of released vdmaheap_cred objects

Definition at line 253 of file vdmaheap_dev.h.

8.182.2.34 stats

```
struct { ... } vdmaheap_device::stats
```

Statistics

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_dev.h

8.183 vdmaheap_device.stats Struct Reference

Data Fields

- atomic_t op_client_create
- atomic_t op_client_create_nok
- atomic_t op_client_create_exited
- atomic_t op_client_destroy
- atomic_t op_client_destroy_exited
- atomic_t op_unexport
- atomic_t op_unexport_nok
- atomic_t op_unexport_exited
- atomic_t op_export
- atomic_t op_export_nok
- atomic_t op_export_exited
- atomic_t op_import
- atomic_t op_import_nok
- atomic_t op_import_exited
- atomic_t op_release
- atomic_t op_release_nok
- atomic_t op_release_exited
- atomic_t op_peer_release
- atomic_t op_peer_release_nok
- atomic_t op_peer_release_exited
- atomic_t op_peer_import
- atomic_t op_peer_import_nok
- atomic_t op_peer_import_exited
- atomic_t vbuf_reuse
- atomic_t vbuf_create
- atomic_t dmabuf_import
- atomic_t dmabuf_import_reused
- atomic_t dmabuf_import_created
- atomic_t dmabuf_import_cache
- atomic_t dmabuf_import_nok
- atomic_t dmabuf_import_sent
- atomic_t dmabuf_release_alive
- atomic_t dmabuf_release_cacheable
- atomic_t dmabuf_release_cached
- atomic_t dmabuf_release_evicted
- atomic_t dma_buf_attach
- atomic_t dma_buf_detach
- atomic_t sg_table_vmalloc
- atomic_t sg_table_vfree
- atomic_t vdmaheap_sg_alloc
- atomic_t vdmaheap_sg_free
- struct avg_stats buf_per_call
- struct avg_stats buf_per_rsp
- struct avg_stats reass_rsp_size
- struct avg_stats rsp_size
- struct avg_stats frag_per_rsp

8.183.1 Detailed Description

Statistics

Definition at line 255 of file vdmaheap_dev.h.

8.183.2 Field Documentation

8.183.2.1 op_client_create

number of entries in vdmaheap_client_create()

8.183.2.2 op_client_create_nok

number of failures for vdmaheap_client_create()

8.183.2.3 op_client_create_exited

number of exits from vdmaheap_client_create()

8.183.2.4 op_client_destroy

number of entries in vdmaheap_client_destroy()

8.183.2.5 op_client_destroy_exited

number of exits in vdmaheap_client_destroy()

8.183.2.6 op_unexport

number of entries in xxx_unexport()

8.183.2.7 op_unexport_nok

number of failures for xxx_unexport()

8.183.2.8 op_unexport_exited

number of exits in xxx_unexport()

8.183.2.9 op_export

number of entries in xxx_export()

8.183.2.10 op_export_nok

number of failures for xxx_export()

8.183.2.11 op_export_exited

number of exits in xxx_export()

8.183.2.12 op_import

number of entries in xxx_import()

8.183.2.13 op_import_nok

number of failures for xxx_import()

8.183.2.14 op_import_exited

number of exits from xxx_import()

8.183.2.15 op_release

number of entries in vdmaheap_dmabuf_release()

8.183.2.16 op_release_nok

number of failures for vdmaheap_dmabuf_release()

8.183.2.17 op_release_exited

number of exits from vdmaheap_dmabuf_release()

8.183.2.18 op_peer_release

number of entries in vdmaheap_vbuf_req_release()

8.183.2.19 op_peer_release_nok

number of failures for vdmaheap_vbuf_req_release()

8.183.2.20 op_peer_release_exited

number of exits in vdmaheap_vbuf_req_release()

8.183.2.21 op_peer_import

number of entries in vdmaheap_vbuf_req_import() / vdmaheap_vbuf_rsp_import()

8.183.2.22 op_peer_import_nok

number of failures for vdmaheap_vbuf_req_import() / vdmaheap_vbuf_rsp_import()

8.183.2.23 op_peer_import_exited

number of exits from vdmaheap_vbuf_req_import() / vdmaheap_vbuf_rsp_import()

8.183.2.24 vbuf_reuse

number of xxx_export() that reuse an already existing vdmaheap_export object

8.183.2.25 vbuf_create

number of xxx_export() that create a new vdmaheap_export object

8.183.2.26 dmabuf_import

number of imported dmabufs

8.183.2.27 dmabuf_import_reused

number of imported dmabufs that are reused as is (i.e. no need to re-build them)

8.183.2.28 dmabuf_import_created

number of imported dmabufs that are created from scratch (i.e. layout needed)

8.183.2.29 dmabuf_import_cache

number of imported dmabufs that are re-built locally (i.e. cached or dying dmabufs)

8.183.2.30 dmabuf_import_nok

number of failed imported dmabufs

8.183.2.31 dmabuf_import_sent

number of imported dmabufs that are sent (i.e. layout needed or cred. check)

8.183.2.32 dmabuf_release_alive

number of imported dmabufs that collided with a release

8.183.2.33 dmabuf_release_cacheable

number of released dmabufs that are cacheable (i.e. with cur_grace $>$ 0)

8.183.2.34 dmabuf_release_cached

number of released dmabufs that are put in the cache

8.183.2.35 dmabuf_release_evicted

number of dmabufs that evicted from the cache after an lru overflow

8.183.2.36 dma_buf_attach

number of dmabuf_attach() called

8.183.2.37 dma_buf_detach

number of dmabuf_deattach() called

8.183.2.38 sg_table_vmalloc

number of allocated sg_table

8.183.2.39 sg_table_vfree

number of freed sg_table

8.183.2.40 vdmaheap_sg_alloc

number of allocated vdmaheap_sg

8.183.2.41 vdmaheap_sg_free

number of freed vdmaheap_sg

8.183.2.42 buf_per_call

number of dmabufs per import() call

8.183.2.43 buf_per_rsp

number of dmabufs per import response message

8.183.2.44 reass_rsp_size

size of reassembled import responses

8.183.2.45 rsp_size

size of import responses messages

8.183.2.46 frag_per_rsp

number of fragments per fragmented response

The documentation for this struct was generated from the following files:

8.184 vdmaheap_dma_buf_ops Struct Reference

8.184.1 Detailed Description

Definition at line 27 of file vdmaheap_dmabuf.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_dmabuf.h

8.185 vdmaheap_heap Struct Reference

8.185.1 Detailed Description

Definition at line 29 of file vdmaheap_heap.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_heap.h

8.186 vdmaheap_import Struct Reference

(a.k.a. vimp) represents the virtual dmabuf (i.e. the dmabuf re-built on the importer's side).

Data Fields

- struct kref kref
- struct mutex lock
- struct vdmaheap_completion completion
- int errno
- struct vdmaheap_client ∗ client
- vdmaheap_buffer_gid_t gid
- struct dma_buf ∗ dmabuf
- struct sg_table ∗ sg_table
- struct vdmaheap_sg ∗ sg
- unsigned int sg_size
- size_t bsize
- unsigned int size
- unsigned long flags
- unsigned long heap_flags
- bool invalid
- bool pending
- unsigned long grace_cur
- unsigned long grace_max
- struct delayed_work evict_dwork
- struct list_head lru_link
- bool is_in_lru

8.186.1 Detailed Description

(a.k.a. vimp) represents the virtual dmabuf (i.e. the dmabuf re-built on the importer's side).

Definition at line 56 of file vdmaheap_import.c.

8.186.2 Field Documentation

8.186.2.1 kref

```
struct kref vdmaheap_import::kref
```

reference counter held by the dmabuf and the vimp_tree

Definition at line 58 of file vdmaheap_import.c.

8.186.2.2 lock

```
struct mutex vdmaheap_import::lock
```

protects this data structure against concurrent accesses

Definition at line 60 of file vdmaheap_import.c.

8.186.2.3 completion

```
struct vdmaheap_completion vdmaheap_import::completion
```

concurrent importers are blocked on this until the vimp import is completed

Definition at line 62 of file vdmaheap_import.c.

8.186.2.4 errno

```
int vdmaheap_import::errno
```

indicates the completion status to a concurrent importer

Definition at line 64 of file vdmaheap_import.c.

8.186.2.5 client

```
struct vdmaheap_client* vdmaheap_import::client
```

importer client (the first one)

Definition at line 66 of file vdmaheap_import.c.

8.186.2.6 gid

`vdmaheap_buffer_gid_t vdmaheap_import::gid`

Global buffer IDentifier (VM id + local buffer id)

Definition at line 68 of file vdmaheap_import.c.

8.186.2.7 dmabuf

`struct dma_buf* vdmaheap_import::dmabuf`

points to the virtual dma_buf object (i.e. rebuilt on the importer's side

Definition at line 70 of file vdmaheap_import.c.

8.186.2.8 sg_table

`struct sg_table* vdmaheap_import::sg_table`

the kernel object for the dmabuf memory layout

Definition at line 72 of file vdmaheap_import.c.

8.186.2.9 sg

`struct vdmaheap_sg* vdmaheap_import::sg`

the vdmaheap object for the dmabuf memory layout serialized from the sg_table

Definition at line 74 of file vdmaheap_import.c.

8.186.2.10 sg_size

`unsigned int vdmaheap_import::sg_size`

size of the sg_table

Definition at line 76 of file vdmaheap_import.c.

8.186.2.11 bsize

```
size_t vdmaheap_import::bsize
```

the buffer size in bytes

Definition at line 78 of file vdmaheap_import.c.

8.186.2.12 size

```
unsigned int vdmaheap_import::size
```

the dmabuf size in bytes

Definition at line 80 of file vdmaheap_import.c.

8.186.2.13 flags

```
unsigned long vdmaheap_import::flags
```

flags of the original ion_buffer object

Definition at line 82 of file vdmaheap_import.c.

8.186.2.14 heap_flags

```
unsigned long vdmaheap_import::heap_flags
```

flags of the heap the original dmabuf was allocated in

Definition at line 84 of file vdmaheap_import.c.

8.186.2.15 invalid

```
bool vdmaheap_import::invalid
```

indicates that the vimp is invalid, must be closed by the importer client

Definition at line 86 of file vdmaheap_import.c.

8.186.2.16 pending

```
bool vdmaheap_import::pending
```

indicates that the vimp is released, but the release request is still pending

Definition at line 88 of file vdmaheap_import.c.

8.186.2.17 grace_cur

```
unsigned long vdmaheap_import::grace_cur
```

current grace delay, will be used when the dmabuf will be put in the cache

Definition at line 93 of file vdmaheap_import.c.

8.186.2.18 grace_max

```
unsigned long vdmaheap_import::grace_max
```

maximal grace delay the maximal delay allowed by exporters clients

Definition at line 95 of file vdmaheap_import.c.

8.186.2.19 evict_dwork

```
struct delayed_work vdmaheap_import::evict_dwork
```

work to be scheduled upon each dmabuf release

Definition at line 97 of file vdmaheap_import.c.

8.186.2.20 lru_link

```
struct list_head vdmaheap_import::lru_link
```

node for the vdmaheap_device lru list

Definition at line 99 of file vdmaheap_import.c.

8.186.2.21 is_in_lru

`bool vdmaheap_import::is_in_lru`

indicates if the vimp is in the lru (i.e. in the cache)

Definition at line 101 of file vdmaheap_import.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_import.c

8.187 vdmaheap_params Struct Reference

8.187.1 Detailed Description

Definition at line 148 of file vdmaheap_dev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_dev.h

8.188 vdmaheap_req_close Struct Reference

8.188.1 Detailed Description

Definition at line 92 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.189 vdmaheap_req_import_mult Struct Reference

8.189.1 Detailed Description

Definition at line 100 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.190 vdmaheap_req_open Struct Reference

8.190.1 Detailed Description

Definition at line 75 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.191 vdmaheap_req_release Struct Reference

8.191.1 Detailed Description

Definition at line 129 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.192 vdmaheap_rsp_close Struct Reference

8.192.1 Detailed Description

Definition at line 96 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.193 vdmaheap_rsp_import_mult Struct Reference

8.193.1 Detailed Description

Definition at line 118 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.194 vdmaheap_rsp_open Struct Reference

8.194.1 Detailed Description

Definition at line 83 of file vdmaheap_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.c

8.195 vdmaheap_sg Struct Reference

vdmaheap data structure for a dmabuf memory layout.

```
#include <vdmaheap_dev.h>
```

Data Fields

- unsigned int size
- unsigned int count
- NkMemMap rgn [ ]

8.195.1 Detailed Description

vdmaheap data structure for a dmabuf memory layout.

Definition at line 374 of file vdmaheap_dev.h.

8.195.2 Field Documentation

8.195.2.1 size

```
unsigned int vdmaheap_sg::size
```

buffer size == sum of rgn[].length

Definition at line 376 of file vdmaheap_dev.h.

8.195.2.2 count

```
unsigned int vdmaheap_sg::count
```

number of regions in rgn[] table

Definition at line 378 of file vdmaheap_dev.h.

8.195.2.3 rgn

```
NkMemMap vdmaheap_sg::rgn[ ]
```

array of count regions

Definition at line 380 of file vdmaheap_dev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_dev.h

8.196 VEth Struct Reference

8.196.1 Detailed Description

Definition at line 246 of file veth.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/veth.c

8.197 VEth.local Struct Reference

8.197.1 Detailed Description

Definition at line 256 of file veth.c.

The documentation for this struct was generated from the following files:

8.198 VEth.peer Struct Reference

8.198.1 Detailed Description

Definition at line 263 of file veth.c.

The documentation for this struct was generated from the following files:

8.199 VEthRingDesc Struct Reference

8.199.1 Detailed Description

Definition at line 139 of file veth.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/veth.c

8.200 VEthSlotDesc Struct Reference

8.200.1 Detailed Description

Definition at line 146 of file veth.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/veth.c

8.201 vevdev_identity_t Struct Reference

8.201.1 Detailed Description

Definition at line 83 of file vevdev_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev_common.h

8.202 vevdev_pip_request Struct Reference

8.202.1 Detailed Description

Definition at line 24 of file vevdev-ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev-ioctl.h

8.203 vevdev_pip_request.u Union Reference

8.203.1 Detailed Description

Definition at line 26 of file vevdev-ioctl.h.

The documentation for this union was generated from the following files:

8.204 vfence2_client Struct Reference

8.204.1 Detailed Description

Definition at line 4284 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.205 vfence2_client_errors Struct Reference

8.205.1 Detailed Description

Definition at line 4231 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.206 vfence2_client_stats Struct Reference

8.206.1 Detailed Description

Definition at line 4165 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.207 vfence2_client_warnings Struct Reference

8.207.1 Detailed Description

Definition at line 4201 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.208 vfence2_connection Struct Reference

8.208.1 Detailed Description

Definition at line 4268 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.209 vfence2_connection_errors Struct Reference

8.209.1 Detailed Description

Definition at line 4123 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.210 vfence2_device Struct Reference

8.210.1 Detailed Description

Definition at line 4299 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.211 vfence2_device.errors Struct Reference

8.211.1 Detailed Description

Definition at line 4321 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.212 vfence2_device.stats Struct Reference

8.212.1 Detailed Description

Definition at line 4311 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.213 vfence2_device.warnings Struct Reference

8.213.1 Detailed Description

Definition at line 4316 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.214 VfenceCltCtrl Struct Reference

8.214.1 Detailed Description

Definition at line 297 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.215 VfenceCltDev Struct Reference

8.215.1 Detailed Description

Definition at line 291 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.216 VfenceCltDev.errors Struct Reference

8.216.1 Detailed Description

Definition at line 603 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.217 VfenceCltDev.stats Struct Reference

8.217.1 Detailed Description

Definition at line 596 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.218 VfenceCltDev.warnings Struct Reference

8.218.1 Detailed Description

Definition at line 600 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.219 VfenceCltFence Struct Reference

8.219.1 Detailed Description

Definition at line 302 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.220 VfenceCltFenceLink Struct Reference

8.220.1 Detailed Description

Definition at line 314 of file vfence.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h

8.221 VfenceCltFile Struct Reference

8.221.1 Detailed Description

Definition at line 331 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.222 VfenceCltFile.__unnamed__ Union Reference

8.222.1 Detailed Description

Definition at line 335 of file vfence.h.

The documentation for this union was generated from the following files:

8.223 VfenceCltFile.__unnamed__ Union Reference

8.223.1 Detailed Description

Definition at line 335 of file vfence.h.

The documentation for this union was generated from the following files:

8.224 VfenceCltFileAlloc Struct Reference

8.224.1 Detailed Description

Definition at line 285 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.225 VfenceCltIFOps Struct Reference

8.225.1 Detailed Description

Definition at line 354 of file vfence.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h

8.226 VfenceCltNode Struct Reference

8.226.1 Detailed Description

Definition at line 322 of file vfence.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h

8.227 VfenceCltRegisterOp Struct Reference

8.227.1 Detailed Description

Definition at line 36 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.228 VfenceCltUnregisterOp Struct Reference

8.228.1 Detailed Description

Definition at line 51 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.229 VfenceCtrlInfo Struct Reference

8.229.1 Detailed Description

Definition at line 29 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.230 VfenceDev Struct Reference

8.230.1 Detailed Description

Definition at line 517 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.231 VfenceDev.__unnamed__ Union Reference

8.231.1 Detailed Description

Definition at line 788 of file vfence2.c.

The documentation for this union was generated from the following files:

8.232 VfenceDev.__unnamed__ Union Reference

8.232.1 Detailed Description

Definition at line 788 of file vfence2.c.

The documentation for this union was generated from the following files:

8.233 VfenceDevOps Struct Reference

8.233.1 Detailed Description

Definition at line 31 of file vfence-drv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/vfence-drv.h

8.234 VfenceDrv Struct Reference

8.234.1 Detailed Description

Definition at line 48 of file vfence-drv.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/vfence-drv.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.235 VfenceFile Struct Reference

8.235.1 Detailed Description

Definition at line 532 of file vfence.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h

8.236 VfenceFile.__unnamed__ Union Reference

8.236.1 Detailed Description

Definition at line 533 of file vfence.h.

The documentation for this union was generated from the following files:

8.237 VfenceFlags Struct Reference

8.237.1 Detailed Description

Definition at line 45 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-proto.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.238 VfenceFlags.__unnamed__ Union Reference

8.238.1 Detailed Description

Definition at line 394 of file vfence2.c.

The documentation for this union was generated from the following files:

8.239 VfenceFlags.__unnamed__ Union Reference

8.239.1 Detailed Description

Definition at line 394 of file vfence2.c.

The documentation for this union was generated from the following files:

8.240 VfenceFlags.__unnamed__.__unnamed__ Struct Reference

8.240.1 Detailed Description

Definition at line 48 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

8.241 VfenceFlags.__unnamed__.__unnamed__ Struct Reference

8.241.1 Detailed Description

Definition at line 48 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

8.242 VfenceGenDev Struct Reference

8.242.1 Detailed Description

Definition at line 81 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.243 VfenceGenFile Struct Reference

8.243.1 Detailed Description

Definition at line 120 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.244 VfenceIdGetOp Struct Reference

8.244.1 Detailed Description

Definition at line 43 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.245 VfenceIFDev Struct Reference

8.245.1 Detailed Description

Definition at line 65 of file vfence-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-os.h

8.246 VfenceIFDev.clt Struct Reference

8.246.1 Detailed Description

Definition at line 67 of file vfence-os.h.

The documentation for this struct was generated from the following files:

8.247 VfenceIFDrv Struct Reference

8.247.1 Detailed Description

Definition at line 61 of file vfence-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-os.h

8.248 VfenceIFFile Struct Reference

8.248.1 Detailed Description

Definition at line 72 of file vfence-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-os.h

8.249 VfenceIFFile.__unnamed__ Union Reference

8.249.1 Detailed Description

Definition at line 74 of file vfence-os.h.

The documentation for this union was generated from the following files:

8.250 VfenceIFFile.__unnamed__.cltFence Struct Reference

8.250.1 Detailed Description

Definition at line 75 of file vfence-os.h.

The documentation for this struct was generated from the following files:

8.251 VfenceIFFile.__unnamed__.srvFence Struct Reference

8.251.1 Detailed Description

Definition at line 79 of file vfence-os.h.

The documentation for this struct was generated from the following files:

8.252 VfenceIFFile.__unnamed__.srvFence.__unnamed__ Union Reference

8.252.1 Detailed Description

Definition at line 80 of file vfence-os.h.

The documentation for this union was generated from the following files:

8.253 VfenceIFOps Struct Reference

8.253.1 Detailed Description

Definition at line 525 of file vfence.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h

8.254 VfenceIFOps.__unnamed__ Union Reference

8.254.1 Detailed Description

Definition at line 526 of file vfence.h.

The documentation for this union was generated from the following files:

8.255 VfenceInfo Struct Reference

8.255.1 Detailed Description

Definition at line 48 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/common.h

8.256 VfenceInfoData Struct Reference

8.256.1 Detailed Description

Definition at line 86 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-proto.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.257 VfenceNativeGetOp Struct Reference

8.257.1 Detailed Description

Definition at line 73 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.258 VfenceParentContext Struct Reference

8.258.1 Detailed Description

Definition at line 19 of file vfence-drv-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/vfence-drv-os.h

8.259 VfencePmemLayout Struct Reference

8.259.1 Detailed Description

Definition at line 164 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-proto.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.260 VfenceProfOp Struct Reference

8.260.1 Detailed Description

Definition at line 82 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.261 VfenceProtoInfo Struct Reference

8.261.1 Detailed Description

Definition at line 131 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-proto.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.262 VfenceProtoInfo.__unnamed__ Union Reference

8.262.1 Detailed Description

Definition at line 473 of file vfence2.c.

The documentation for this union was generated from the following files:

8.263 VfenceProtoInfo.__unnamed__ Union Reference

8.263.1 Detailed Description

Definition at line 473 of file vfence2.c.

The documentation for this union was generated from the following files:

8.264 VfenceProtoInfo.__unnamed__.__unnamed__ Struct Reference

8.264.1 Detailed Description

Definition at line 474 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.265 VfenceProtoInfo.__unnamed__.__unnamed__ Struct Reference

8.265.1 Detailed Description

Definition at line 474 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.266 VfencePtInfo Struct Reference

8.266.1 Detailed Description

Definition at line 80 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-proto.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.267 VfenceSlot Struct Reference

8.267.1 Detailed Description

Definition at line 92 of file vfence-proto.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence-proto.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.268 VfenceSlot.__unnamed__ Union Reference

8.268.1 Detailed Description

Definition at line 101 of file vfence-proto.h.

The documentation for this union was generated from the following files:

8.269 VfenceSlot.__unnamed__ Union Reference

8.269.1 Detailed Description

Definition at line 101 of file vfence-proto.h.

The documentation for this union was generated from the following files:

8.270 VfenceSlot.__unnamed__ Union Reference

8.270.1 Detailed Description

Definition at line 101 of file vfence-proto.h.

The documentation for this union was generated from the following files:

8.271 VfenceSlot.__unnamed__ Union Reference

8.271.1 Detailed Description

Definition at line 101 of file vfence-proto.h.

The documentation for this union was generated from the following files:

8.272 VfenceSlot.__unnamed__ Union Reference

8.272.1 Detailed Description

Definition at line 101 of file vfence-proto.h.

The documentation for this union was generated from the following files:

8.273 VfenceSrvCtrl Struct Reference

8.273.1 Detailed Description

Definition at line 415 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.274 VfenceSrvCtrlReleaseCtx Struct Reference

8.274.1 Detailed Description

Definition at line 2186 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.275 VfenceSrvDev Struct Reference

8.275.1 Detailed Description

Definition at line 407 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.276 VfenceSrvDev.errors Struct Reference

8.276.1 Detailed Description

Definition at line 702 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.277 VfenceSrvDev.stats Struct Reference

8.277.1 Detailed Description

Definition at line 694 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.278 VfenceSrvFence Struct Reference

8.278.1 Detailed Description

Definition at line 422 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.279 VfenceSrvFile Struct Reference

8.279.1 Detailed Description

Definition at line 440 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.280 VfenceSrvFile.__unnamed__ Union Reference

8.280.1 Detailed Description

Definition at line 444 of file vfence.h.

The documentation for this union was generated from the following files:

8.281 VfenceSrvFile.__unnamed__ Union Reference

8.281.1 Detailed Description

Definition at line 444 of file vfence.h.

The documentation for this union was generated from the following files:

8.282 VfenceSrvFile.srvFence Struct Reference

8.282.1 Detailed Description

Definition at line 738 of file vfence2.c.

The documentation for this struct was generated from the following files:

8.283 VfenceSrvIdxAlloc Struct Reference

8.283.1 Detailed Description

Definition at line 398 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.284 VfenceSrvIdxItem Struct Reference

8.284.1 Detailed Description

Definition at line 392 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.285 VfenceSrvIFOps Struct Reference

8.285.1 Detailed Description

Definition at line 456 of file vfence.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h

8.286 VfenceSrvRegisterOp Struct Reference

8.286.1 Detailed Description

Definition at line 59 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.287 VfenceSrvUnregisterOp Struct Reference

8.287.1 Detailed Description

Definition at line 66 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/ioctl.h

8.288 VfenceStat Struct Reference

8.288.1 Detailed Description

Definition at line 117 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence/common.h

8.289 VfenceStats Struct Reference

8.289.1 Detailed Description

Definition at line 135 of file vfence.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h

8.290 VfenceStats.__unnamed__ Union Reference

8.290.1 Detailed Description

Definition at line 136 of file vfence.h.

The documentation for this union was generated from the following files:

8.291 VfenceStats.__unnamed__.__unnamed__ Struct Reference

8.291.1 Detailed Description

Definition at line 137 of file vfence.h.

The documentation for this struct was generated from the following files:

8.292 VfenceXirqMap Struct Reference

8.292.1 Detailed Description

Definition at line 75 of file vfence.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence.h
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.293 vg2d Struct Reference

8.293.1 Detailed Description

Definition at line 100 of file vg2d-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vg2d-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vg2d-fe.c

8.294 vg2d.secure Struct Reference

8.294.1 Detailed Description

Definition at line 121 of file vg2d-be.c.

The documentation for this struct was generated from the following files:

8.295 vg2d_ipc Union Reference

8.295.1 Detailed Description

Definition at line 125 of file vg2d_common.h.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vg2d_common.h

8.296 vg2d_req Struct Reference

8.296.1 Detailed Description

Definition at line 76 of file vg2d_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vg2d_common.h

8.297 vg2d_req_ioc_perf Struct Reference

8.297.1 Detailed Description

Definition at line 112 of file vg2d_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vg2d_common.h

8.298 vg2d_req_ioc_prio Struct Reference

8.298.1 Detailed Description

Definition at line 107 of file vg2d_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vg2d_common.h

8.299 vg2d_req_ioc_proc Struct Reference

8.299.1 Detailed Description

Definition at line 84 of file vg2d_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vg2d_common.h

8.300 vg2d_res Struct Reference

8.300.1 Detailed Description

Definition at line 117 of file vg2d_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vg2d_common.h

8.301 vgki_driver Struct Reference

Data Structures

- struct vgki_driver_errors
- struct vgki_driver_kernel
- struct vgki_driver_stats
- struct vgki_driver_user

8.301.1 Detailed Description

Definition at line 112 of file vgki.c.

8.301.2 Data Structure Documentation

8.301.2.1 struct vgki_driver::vgki_driver_errors

Definition at line 146 of file vgki.c.

8.301.2.2 struct vgki_driver::vgki_driver_kernel

Definition at line 121 of file vgki.c.

8.301.2.3 struct vgki_driver::vgki_driver_stats

Definition at line 130 of file vgki.c.

8.301.2.4 struct vgki_driver::vgki_driver_user

Definition at line 117 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.302 vgki_file Struct Reference

8.302.1 Detailed Description

Definition at line 105 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.303 vgki_ioc_version Struct Reference

8.303.1 Detailed Description

Definition at line 53 of file vgki-uapi.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-uapi.h

8.304 vgki_ioc_wait Struct Reference

8.304.1 Detailed Description

Definition at line 66 of file vgki-uapi.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-uapi.h

8.305 vgki_ioc_wait_io Struct Reference

8.305.1 Detailed Description

Definition at line 57 of file vgki-uapi.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-uapi.h

8.306 vgki_kernel Struct Reference

Data Structures

- union vgki_kernel_args

8.306.1 Detailed Description

Definition at line 216 of file vgki.c.

8.306.2 Data Structure Documentation

8.306.2.1 union vgki_kernel::vgki_kernel_args

Definition at line 225 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.307 vgki_kernel_close Struct Reference

8.307.1 Detailed Description

Definition at line 193 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.308 vgki_kernel_kernel_thread Struct Reference

8.308.1 Detailed Description

Definition at line 184 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.309 vgki_kernel_mmap Struct Reference

8.309.1 Detailed Description

Definition at line 200 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.310 vgki_kernel_munmap Struct Reference

8.310.1 Detailed Description

Definition at line 208 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.311 vgki_kernel_open Struct Reference

8.311.1 Detailed Description

Definition at line 174 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.312 vgki_open_out Struct Reference

8.312.1 Detailed Description

Definition at line 47 of file vgki-uapi.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-uapi.h

8.313 vgki_starter Struct Reference

8.313.1 Detailed Description

Definition at line 63 of file vgki-kapi.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-kapi.h

8.314 vgki_thread Struct Reference

Thread library allowing clean termination of VGKI threads.

Data Fields

struct vgki_thread

- int(∗ func)(void ∗data)
  Thread start routine.
- void ∗ data
  Argument to start function.
- const char ∗ name
  Name for the thread.
- pid_t pid
  Thread identifier of the thread.
- struct completion completion
  Object allowing to safely await termination of thread.
- long exit_code
  Exit value of the thread start routine.

8.314.1 Detailed Description

Thread library allowing clean termination of VGKI threads.

Definition at line 42 of file vgki-kapi.h.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-kapi.h
- linux-drivers-vl-3.18/linux-kernel-vdrivers/doc/vdrivers_4D/api-vgki.dox

8.315 vgki_user Struct Reference

Data Structures

- union vgki_user_u

8.315.1 Detailed Description

Definition at line 234 of file vgki.c.

8.315.2 Data Structure Documentation

8.315.2.1 union vgki_user::vgki_user_u

Definition at line 239 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.316 vgki_user_call Struct Reference

8.316.1 Detailed Description

Definition at line 247 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.317 vgki_work Struct Reference

8.317.1 Detailed Description

Definition at line 89 of file vgki-kapi.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-kapi.h

8.318 vgki_workqueue Struct Reference

8.318.1 Detailed Description

Definition at line 1971 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.319 vgki_workqueue.stats Struct Reference

8.319.1 Detailed Description

Definition at line 1996 of file vgki.c.

The documentation for this struct was generated from the following files:

8.320 vgki_workqueue.warnings Struct Reference

8.320.1 Detailed Description

Definition at line 1991 of file vgki.c.

The documentation for this struct was generated from the following files:

8.321 vgki_workqueue.works Struct Reference

8.321.1 Detailed Description

Definition at line 1987 of file vgki.c.

The documentation for this struct was generated from the following files:

8.322 vgki_wq_thread Struct Reference

8.322.1 Detailed Description

Definition at line 1947 of file vgki.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki.c

8.323 vgki_wq_thread.stats Struct Reference

8.323.1 Detailed Description

Definition at line 1963 of file vgki.c.

The documentation for this struct was generated from the following files:

8.324 vgkit_thread Struct Reference

8.324.1 Detailed Description

Definition at line 77 of file vgki-test.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vgki-test.c

8.325 vi2c_addr_t Struct Reference

8.325.1 Detailed Description

Definition at line 62 of file vi2c_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vi2c_common.h

8.326 vi2c_cmd_t Struct Reference

8.326.1 Detailed Description

Definition at line 55 of file vi2c_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vi2c_common.h

8.327 vi2c_link_t Struct Reference

8.327.1 Detailed Description

Definition at line 45 of file vi2c-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vi2c-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vi2c-fe.c

8.328 vi2c_msg_t Struct Reference

8.328.1 Detailed Description

Definition at line 67 of file vi2c_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vi2c_common.h

8.329 vi2c_op_t Struct Reference

8.329.1 Detailed Description

Definition at line 74 of file vi2c_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vi2c_common.h

8.330 vi2c_transfer_t Struct Reference

8.330.1 Detailed Description

Definition at line 80 of file vi2c_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vi2c_common.h

8.331 Vldev Struct Reference

8.331.1 Detailed Description

Definition at line 56 of file vevdev_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev_common.h

8.332 vinfo_file Struct Reference

8.332.1 Detailed Description

Definition at line 2862 of file vinfo.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vinfo.c

8.333 vion_buffer Struct Reference

(a.k.a. vbuf) represents the original dmabuf (i.e. the exported dmabuf).

Data Fields

- struct list_head vbuf_node
- struct kref kref
- struct vion_completion completion
- int errno
- struct mutex cred_list_lock
- vion_buffer_gid_t gid
- struct dma_buf ∗ dmabuf
- struct sg_table ∗ sg_table
- struct vion_sg ∗ sg
- unsigned int sg_size
- struct vion_client ∗ client
- struct list_head cred_list
- nku32_f granted [NK_OS_LIMIT]
- unsigned long heap_flags
- unsigned long grace_max

8.333.1 Detailed Description

(a.k.a. vbuf) represents the original dmabuf (i.e. the exported dmabuf).

Definition at line 62 of file vion_export.c.

8.333.2 Field Documentation

8.333.2.1 vbuf_node

`struct list_head vion_buffer::vbuf_node`

node for the vion_device list

Definition at line 64 of file vion_export.c.

8.333.2.2 kref

`struct kref vion_buffer::kref`

reference counter held by vion_cred and the idr_tree

Definition at line 67 of file vion_export.c.

8.333.2.3 completion

`struct vion_completion vion_buffer::completion`

concurrent exporters are blocked on this until the vbuf creation is completed

Definition at line 69 of file vion_export.c.

8.333.2.4 errno

`int vion_buffer::errno`

indicates the completion status to a concurrent exporter

Definition at line 71 of file vion_export.c.

8.333.2.5 cred_list_lock

`struct mutex vion_buffer::cred_list_lock`

protects against concurrent accesses to the cred list

Definition at line 73 of file vion_export.c.

8.333.2.6 gid

`vion_buffer_gid_t vion_buffer::gid`

Global buffer IDentifier (VM id + local buffer id)

Definition at line 75 of file vion_export.c.

8.333.2.7 dmabuf

`struct dma_buf* vion_buffer::dmabuf`

points to the original dma_buf object

Definition at line 77 of file vion_export.c.

8.333.2.8 sg_table

`struct sg_table* vion_buffer::sg_table`

the kernel object for the dmabuf memory layout

Definition at line 80 of file vion_export.c.

8.333.2.9 sg

`struct vion_sg* vion_buffer::sg`

the vion object for the dmabuf memory layout serialized from the sg_table

Definition at line 82 of file vion_export.c.

8.333.2.10 sg_size

`unsigned int vion_buffer::sg_size`

size of the sg_table

Definition at line 84 of file vion_export.c.

8.333.2.11 client

```
struct vion_client* vion_buffer::client
```

exporter client (the first one)

Definition at line 86 of file vion_export.c.

8.333.2.12 cred_list

```
struct list_head vion_buffer::cred_list
```

list of export credentials associated with that buffer

Definition at line 88 of file vion_export.c.

8.333.2.13 granted

```
nku32_f vion_buffer::granted[NK_OS_LIMIT]
```

per-vm grant access flags

Definition at line 90 of file vion_export.c.

8.333.2.14 heap_flags

```
unsigned long vion_buffer::heap_flags
```

flags of the heap the dmabuf was allocated in

Definition at line 92 of file vion_export.c.

8.333.2.15 grace_max

```
unsigned long vion_buffer::grace_max
```

maximal grace delay allowed by the dmabuf exporters clients

Definition at line 94 of file vion_export.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_export.c

8.334 vion_cmd Struct Reference

8.334.1 Detailed Description

Definition at line 63 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.335 vion_cmd_res_release Struct Reference

8.335.1 Detailed Description

Definition at line 134 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.336 vion_completion Struct Reference

8.336.1 Detailed Description

Definition at line 112 of file vion_dev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_dev.h

8.337 vion_device Struct Reference

vion_device data structure.

`#include <vion_dev.h>`

Data Fields

- atomic_t link_state
- wait_queue_head_t link_wq
- struct list_head client_list
- struct list_head vbuf_list
- struct list_head lru_list
- struct list_head be_list
- struct list_head fe_list
- struct list_head dmabuf_ops_list
- struct idr vbuf_idr
- struct radix_tree_root vbuf_tree
- struct radix_tree_root vimp_tree
- struct mutex lock
- struct mutex lru_list_lock
- struct mutex vimp_tree_lock
- struct mutex client_list_lock
- struct mutex vbuf_list_lock
- struct mutex vbuf_tree_lock
- struct mutex vbuf_idr_lock
- struct mutex dmabuf_ops_lock
- struct vion_heap ∗ vion_heap
- unsigned long lru_size_max
- unsigned long lru_size
- unsigned int vrpcs_minsize
- atomic_t vbuf_create_count
- atomic_t vbuf_release_count
- atomic_t client_create_count
- atomic_t client_release_count
- atomic_t vimp_create_count
- atomic_t vimp_release_count
- atomic_t dmabuf_create_count
- atomic_t dmabuf_release_count
- atomic_t cred_create_count
- atomic_t cred_release_count
- struct {
  atomic_t **op_client_create**
  atomic_t **op_client_create_nok**
  atomic_t **op_client_create_exited**
  atomic_t **op_client_destroy**
  atomic_t **op_client_destroy_exited**
  atomic_t **op_unexport**
  atomic_t **op_unexport_nok**
  atomic_t **op_unexport_exited**
  atomic_t **op_export**
  atomic_t **op_export_nok**
  atomic_t **op_export_exited**
  atomic_t **op_import**
  atomic_t **op_import_nok**
  atomic_t **op_import_exited**
  atomic_t **op_release**
  atomic_t **op_release_nok**
  atomic_t **op_release_exited**
  atomic_t **op_peer_release**
  atomic_t **op_peer_release_nok**
  atomic_t **op_peer_release_exited**

atomic_t **op_peer_import**
atomic_t **op_peer_import_nok**
atomic_t **op_peer_import_exited**
atomic_t **vbuf_reuse**
atomic_t **vbuf_create**
atomic_t **dmabuf_import**
atomic_t **dmabuf_import_reused**
atomic_t **dmabuf_import_created**
atomic_t **dmabuf_import_cache**
atomic_t **dmabuf_import_nok**
atomic_t **dmabuf_import_sent**
atomic_t **dmabuf_release_alive**
atomic_t **dmabuf_release_cacheable**
atomic_t **dmabuf_release_cached**
atomic_t **dmabuf_release_evicted**
atomic_t **dma_buf_attach**
atomic_t **dma_buf_detach**
atomic_t **sg_table_vmalloc**
atomic_t **sg_table_vfree**
atomic_t **vion_sg_alloc**
atomic_t **vion_sg_free**
struct avg_stats **buf_per_call**
struct avg_stats **buf_per_rsp**
struct avg_stats **reass_rsp_size**
struct avg_stats **rsp_size**
struct avg_stats **frag_per_rsp**
} stats

8.337.1 Detailed Description

vion_device data structure.

the top object in the vion hierarchy.

Definition at line 154 of file vion_dev.h.

8.337.2 Field Documentation

8.337.2.1 link_state

```
atomic_t vion_device::link_state
```

indicates the state of all vion links with the peer entities in the form of a bitmap. The bit i indicates the state of the vion link with the peer domain i:

- 1: the link is up.
- 0: the link is down. When a link is reported in the down state, all dmabufs imported from the corresponding domain must be closed by their owners.

Definition at line 168 of file vion_dev.h.

8.337.2.2 link_wq

`wait_queue_head_t vion_device::link_wq`

kernel wait_queue object associated with the link_state

Definition at line 170 of file vion_dev.h.

8.337.2.3 client_list

`struct list_head vion_device::client_list`

references all the vion_device vion_client objects

Definition at line 174 of file vion_dev.h.

8.337.2.4 vbuf_list

`struct list_head vion_device::vbuf_list`

references all the vion_device vion_buffer objects

Definition at line 176 of file vion_dev.h.

8.337.2.5 lru_list

`struct list_head vion_device::lru_list`

references all the vion_device vion_import objects (cache)

Definition at line 178 of file vion_dev.h.

8.337.2.6 be_list

`struct list_head vion_device::be_list`

references all the vion_device vion_be objects

Definition at line 180 of file vion_dev.h.

8.337.2.7 fe_list

```
struct list_head vion_device::fe_list
```

references all the vion_device vion_fe objects

Definition at line 182 of file vion_dev.h.

8.337.2.8 dmabuf_ops_list

```
struct list_head vion_device::dmabuf_ops_list
```

references all the vion_device templates for the caught dmabufs

Definition at line 184 of file vion_dev.h.

8.337.2.9 vbuf_idr

```
struct idr vion_device::vbuf_idr
```

references by their GID the vion_device vion_buffer objects

Definition at line 186 of file vion_dev.h.

8.337.2.10 vbuf_tree

```
struct radix_tree_root vion_device::vbuf_tree
```

references by their dmabuf address the vion_device vion_buffer objects

Definition at line 188 of file vion_dev.h.

8.337.2.11 vimp_tree

```
struct radix_tree_root vion_device::vimp_tree
```

references by their GID the vion_device vion_import objects

Definition at line 190 of file vion_dev.h.

8.337.2.12 lock

```
struct mutex vion_device::lock
```

protects this data structure against concurrent accesses

Definition at line 194 of file vion_dev.h.

8.337.2.13 lru_list_lock

```
struct mutex vion_device::lru_list_lock
```

protects the lru list against concurrent accesses

Definition at line 196 of file vion_dev.h.

8.337.2.14 vimp_tree_lock

```
struct mutex vion_device::vimp_tree_lock
```

protects vimp_tree against concurrent accesses

Definition at line 198 of file vion_dev.h.

8.337.2.15 client_list_lock

```
struct mutex vion_device::client_list_lock
```

protects client_list against concurrent accesses

Definition at line 200 of file vion_dev.h.

8.337.2.16 vbuf_list_lock

```
struct mutex vion_device::vbuf_list_lock
```

protects vbuf_list against concurrent accesses

Definition at line 202 of file vion_dev.h.

8.337.2.17 vbuf_tree_lock

```
struct mutex vion_device::vbuf_tree_lock
```

protects vbuf_tree against concurrent accesses

Definition at line 204 of file vion_dev.h.

8.337.2.18 vbuf_idr_lock

```
struct mutex vion_device::vbuf_idr_lock
```

protects vbuf_idr against concurrent accesses

Definition at line 206 of file vion_dev.h.

8.337.2.19 dmabuf_ops_lock

```
struct mutex vion_device::dmabuf_ops_lock
```

protects dmabuf_ops_list against concurrent accesses

Definition at line 208 of file vion_dev.h.

8.337.2.20 vion_heap

```
struct vion_heap* vion_device::vion_heap
```

pseudo heap featuring ops for imported remote buffers

Definition at line 211 of file vion_dev.h.

8.337.2.21 lru_size_max

```
unsigned long vion_device::lru_size_max
```

maximal cumulated size of the dmabufs "in-flight" in the cache

Definition at line 214 of file vion_dev.h.

8.337.2.22 lru_size

```
unsigned long vion_device::lru_size
```

current cumulated size of the dmabufs "in-flight" in the cache

Definition at line 216 of file vion_dev.h.

8.337.2.23 vrpcs_minsize

```
unsigned int vion_device::vrpcs_minsize
```

minimal size for the pmem

Definition at line 220 of file vion_dev.h.

8.337.2.24 vbuf_create_count

```
atomic_t vion_device::vbuf_create_count
```

number of created vion_buffer objects

Definition at line 224 of file vion_dev.h.

8.337.2.25 vbuf_release_count

```
atomic_t vion_device::vbuf_release_count
```

number of released vion_buffer objects

Definition at line 226 of file vion_dev.h.

8.337.2.26 client_create_count

```
atomic_t vion_device::client_create_count
```

number of created vion_client objects

Definition at line 228 of file vion_dev.h.

8.337.2.27 client_release_count

`atomic_t vion_device::client_release_count`

number of released vion_client objects

Definition at line 230 of file vion_dev.h.

8.337.2.28 vimp_create_count

`atomic_t vion_device::vimp_create_count`

number of created vion_import objects

Definition at line 232 of file vion_dev.h.

8.337.2.29 vimp_release_count

`atomic_t vion_device::vimp_release_count`

number of released vion_import objects

Definition at line 234 of file vion_dev.h.

8.337.2.30 dmabuf_create_count

`atomic_t vion_device::dmabuf_create_count`

number of created virtual dmabufs objects

Definition at line 236 of file vion_dev.h.

8.337.2.31 dmabuf_release_count

`atomic_t vion_device::dmabuf_release_count`

number of released virtual dmabufs objects

Definition at line 238 of file vion_dev.h.

8.337.2.32 cred_create_count

```
atomic_t vion_device::cred_create_count
```

number of created vion_cred objects

Definition at line 240 of file vion_dev.h.

8.337.2.33 cred_release_count

```
atomic_t vion_device::cred_release_count
```

number of released vion_cred objects

Definition at line 242 of file vion_dev.h.

8.337.2.34 stats

```
struct { ... } vion_device::stats
```

Statistics

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_dev.h

8.338 vion_device.stats Struct Reference

Data Fields

- atomic_t op_client_create
- atomic_t op_client_create_nok
- atomic_t op_client_create_exited
- atomic_t op_client_destroy
- atomic_t op_client_destroy_exited
- atomic_t op_unexport
- atomic_t op_unexport_nok
- atomic_t op_unexport_exited
- atomic_t op_export
- atomic_t op_export_nok
- atomic_t op_export_exited
- atomic_t op_import
- atomic_t op_import_nok
- atomic_t op_import_exited
- atomic_t op_release
- atomic_t op_release_nok

- atomic_t op_release_exited
- atomic_t op_peer_release
- atomic_t op_peer_release_nok
- atomic_t op_peer_release_exited
- atomic_t op_peer_import
- atomic_t op_peer_import_nok
- atomic_t op_peer_import_exited
- atomic_t vbuf_reuse
- atomic_t vbuf_create
- atomic_t dmabuf_import
- atomic_t dmabuf_import_reused
- atomic_t dmabuf_import_created
- atomic_t dmabuf_import_cache
- atomic_t dmabuf_import_nok
- atomic_t dmabuf_import_sent
- atomic_t dmabuf_release_alive
- atomic_t dmabuf_release_cacheable
- atomic_t dmabuf_release_cached
- atomic_t dmabuf_release_evicted
- atomic_t dma_buf_attach
- atomic_t dma_buf_detach
- atomic_t sg_table_vmalloc
- atomic_t sg_table_vfree
- atomic_t vion_sg_alloc
- atomic_t vion_sg_free
- struct avg_stats buf_per_call
- struct avg_stats buf_per_rsp
- struct avg_stats reass_rsp_size
- struct avg_stats rsp_size
- struct avg_stats frag_per_rsp

8.338.1 Detailed Description

Statistics

Definition at line 244 of file vion_dev.h.

8.338.2 Field Documentation

8.338.2.1 op_client_create

number of entries in vion_client_create()

8.338.2.2 op_client_create_nok

number of failures for vion_client_create()

8.338.2.3 op_client_create_exited

number of exits from vion_client_create()

8.338.2.4 op_client_destroy

number of entries in vion_client_destroy()

8.338.2.5 op_client_destroy_exited

number of exits in vion_client_destroy()

8.338.2.6 op_unexport

number of entries in xxx_unexport()

8.338.2.7 op_unexport_nok

number of failures for xxx_unexport()

8.338.2.8 op_unexport_exited

number of exits in xxx_unexport()

8.338.2.9 op_export

number of entries in xxx_export()

8.338.2.10 op_export_nok

number of failures for xxx_export()

8.338.2.11 op_export_exited

number of exits in xxx_export()

8.338.2.12 op_import

number of entries in xxx_import()

8.338.2.13 op_import_nok

number of failures for xxx_import()

8.338.2.14 op_import_exited

number of exits from xxx_import()

8.338.2.15 op_release

number of entries in vion_dmabuf_release()

8.338.2.16 op_release_nok

number of failures for vion_dmabuf_release()

8.338.2.17 op_release_exited

number of exits from vion_dmabuf_release()

8.338.2.18 op_peer_release

number of entries in vion_vbuf_req_release()

8.338.2.19 op_peer_release_nok

number of failures for vion_vbuf_req_release()

8.338.2.20 op_peer_release_exited

number of exits in vion_vbuf_req_release()

8.338.2.21 op_peer_import

number of entries in vion_vbuf_req_import() / vion_vbuf_rsp_import()

8.338.2.22 op_peer_import_nok

number of failures for vion_vbuf_req_import() / vion_vbuf_rsp_import()

8.338.2.23 op_peer_import_exited

number of exits from vion_vbuf_req_import() / vion_vbuf_rsp_import()

8.338.2.24 vbuf_reuse

number of xxx_export() that reuse an already existing vion_export object

8.338.2.25 vbuf_create

number of xxx_export() that create a new vion_export object

8.338.2.26 dmabuf_import

number of imported dmabufs

8.338.2.27 dmabuf_import_reused

number of imported dmabufs that are reused as is (i.e. no need to re-build them)

8.338.2.28 dmabuf_import_created

number of imported dmabufs that are created from scratch (i.e. layout needed)

8.338.2.29 dmabuf_import_cache

number of imported dmabufs that are re-built locally (i.e. cached or dying dmabufs)

8.338.2.30 dmabuf_import_nok

number of failed imported dmabufs

8.338.2.31 dmabuf_import_sent

number of imported dmabufs that are sent (i.e. layout needed or cred. check)

8.338.2.32 dmabuf_release_alive

number of imported dmabufs that collided with a release

8.338.2.33 dmabuf_release_cacheable

number of released dmabufs that are cacheable (i.e. with cur_grace $>$ 0)

8.338.2.34 dmabuf_release_cached

number of released dmabufs that are put in the cache

8.338.2.35 dmabuf_release_evicted

number of dmabufs that evicted from the cache after an lru overflow

8.338.2.36 dma_buf_attach

number of dmabuf_attach() called

8.338.2.37 dma_buf_detach

number of dmabuf_deattach() called

8.338.2.38 sg_table_vmalloc

number of allocated sg_table

8.338.2.39 sg_table_vfree

number of freed sg_table

8.338.2.40 vion_sg_alloc

number of allocated vion_sg

8.338.2.41 vion_sg_free

number of freed vion_sg

8.338.2.42 buf_per_call

number of dmabufs per import() call

8.338.2.43 buf_per_rsp

number of dmabufs per import response message

8.338.2.44 reass_rsp_size

size of reassembled import responses

8.338.2.45 rsp_size

size of import responses messages

8.338.2.46 frag_per_rsp

number of fragments per fragmented response

The documentation for this struct was generated from the following files:

8.339 vion_dma_buf_ops Struct Reference

8.339.1 Detailed Description

Definition at line 32 of file vion_dmabuf.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_dmabuf.h

8.340 vion_heap Struct Reference

8.340.1 Detailed Description

Definition at line 32 of file vion_heap.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_heap.h

8.341 vion_import Struct Reference

(a.k.a. vimp) represents the virtual dmabuf (i.e. the dmabuf re-built on the importer's side).

Data Fields

- struct kref kref
- struct mutex lock
- struct vion_completion completion
- int errno
- struct vion_client * client
- vion_buffer_gid_t gid
- struct dma_buf * dmabuf
- struct sg_table * sg_table
- struct vion_sg * sg
- unsigned int sg_size
- size_t bsize
- unsigned int size
- unsigned long flags
- unsigned long heap_flags
- bool invalid
- bool pending
- unsigned long grace_cur
- unsigned long grace_max
- struct delayed_work evict_dwork
- struct list_head lru_link
- bool is_in_lru

8.341.1 Detailed Description

(a.k.a. vimp) represents the virtual dmabuf (i.e. the dmabuf re-built on the importer's side).

Definition at line 66 of file vion_import.c.

8.341.2 Field Documentation

8.341.2.1 kref

`struct kref vion_import::kref`

reference counter held by the dmabuf and the vimp_tree

Definition at line 68 of file vion_import.c.

8.341.2.2 lock

`struct mutex vion_import::lock`

protects this data structure against concurrent accesses

Definition at line 70 of file vion_import.c.

8.341.2.3 completion

```
struct vion_completion vion_import::completion
```

concurrent importers are blocked on this until the vimp import is completed

Definition at line 72 of file vion_import.c.

8.341.2.4 errno

```
int vion_import::errno
```

indicates the completion status to a concurrent importer

Definition at line 74 of file vion_import.c.

8.341.2.5 client

```
struct vion_client* vion_import::client
```

importer client (the first one)

Definition at line 76 of file vion_import.c.

8.341.2.6 gid

```
vion_buffer_gid_t vion_import::gid
```

Global buffer IDentifier (VM id + local buffer id)

Definition at line 78 of file vion_import.c.

8.341.2.7 dmabuf

```
struct dma_buf* vion_import::dmabuf
```

points to the virtual dma_buf object (i.e. rebuilt on the importer's side

Definition at line 80 of file vion_import.c.

8.341.2.8 sg_table

```
struct sg_table* vion_import::sg_table
```

the kernel object for the dmabuf memory layout

Definition at line 82 of file vion_import.c.

8.341.2.9 sg

```
struct vion_sg* vion_import::sg
```

the vion object for the dmabuf memory layout serialized from the sg_table

Definition at line 84 of file vion_import.c.

8.341.2.10 sg_size

```
unsigned int vion_import::sg_size
```

size of the sg_table

Definition at line 86 of file vion_import.c.

8.341.2.11 bsize

```
size_t vion_import::bsize
```

the buffer size in bytes

Definition at line 88 of file vion_import.c.

8.341.2.12 size

```
unsigned int vion_import::size
```

the dmabuf size in bytes

Definition at line 90 of file vion_import.c.

8.341.2.13 flags

`unsigned long vion_import::flags`

flags of the original ion_buffer object

Definition at line 92 of file vion_import.c.

8.341.2.14 heap_flags

`unsigned long vion_import::heap_flags`

flags of the heap the original dmabuf was allocated in

Definition at line 94 of file vion_import.c.

8.341.2.15 invalid

`bool vion_import::invalid`

indicates that the vimp is invalid, must be closed by the importer client

Definition at line 96 of file vion_import.c.

8.341.2.16 pending

`bool vion_import::pending`

indicates that the vimp is released, but the release request is still pending

Definition at line 98 of file vion_import.c.

8.341.2.17 grace_cur

`unsigned long vion_import::grace_cur`

current grace delay, will be used when the dmabuf will be put in the cache

Definition at line 103 of file vion_import.c.

8.341.2.18 grace_max

```
unsigned long vion_import::grace_max
```

maximal grace delay the maximal delay allowed by exporters clients

Definition at line 105 of file vion_import.c.

8.341.2.19 evict_dwork

```
struct delayed_work vion_import::evict_dwork
```

work to be scheduled upon each dmabuf release

Definition at line 107 of file vion_import.c.

8.341.2.20 lru_link

```
struct list_head vion_import::lru_link
```

node for the vion_device lru list

Definition at line 109 of file vion_import.c.

8.341.2.21 is_in_lru

```
bool vion_import::is_in_lru
```

indicates if the vimp is in the lru (i.e. in the cache)

Definition at line 111 of file vion_import.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_import.c

8.342 vion_params Struct Reference

8.342.1 Detailed Description

Definition at line 137 of file vion_dev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_dev.h

8.343 vion_req_close Struct Reference

8.343.1 Detailed Description

Definition at line 92 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.344 vion_req_import_mult Struct Reference

8.344.1 Detailed Description

Definition at line 100 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.345 vion_req_open Struct Reference

8.345.1 Detailed Description

Definition at line 75 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.346 vion_req_release Struct Reference

8.346.1 Detailed Description

Definition at line 129 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.347 vion_rsp_close Struct Reference

8.347.1 Detailed Description

Definition at line 96 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.348 vion_rsp_import_mult Struct Reference

8.348.1 Detailed Description

Definition at line 118 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.349 vion_rsp_open Struct Reference

8.349.1 Detailed Description

Definition at line 83 of file vion_vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.c

8.350 vion_sg Struct Reference

vion data structure for a dmabuf memory layout.

```
#include <vion_dev.h>
```

Data Fields

- unsigned int size
- unsigned int count
- NkMemMap rgn [ ]

8.350.1 Detailed Description

vion data structure for a dmabuf memory layout.

Definition at line 363 of file vion_dev.h.

8.350.2 Field Documentation

8.350.2.1 size

```
unsigned int vion_sg::size
```

buffer size == sum of rgn[].length

Definition at line 365 of file vion_dev.h.

8.350.2.2 count

```
unsigned int vion_sg::count
```

number of regions in rgn[] table

Definition at line 367 of file vion_dev.h.

8.350.2.3 rgn

```
NkMemMap vion_sg::rgn[]
```

array of count regions

Definition at line 369 of file vion_dev.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_dev.h

8.351 vipc_cookie_t Union Reference

8.351.1 Detailed Description

Definition at line 77 of file vlx-vipc.c.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vipc.c

8.352 vipc_ctx_t Struct Reference

8.352.1 Detailed Description

Definition at line 86 of file vlx-vipc.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vipc.h

8.353 vipc_list_t Struct Reference

8.353.1 Detailed Description

Definition at line 38 of file vlx-vipc.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vipc.h

8.354 vipc_result_t Struct Reference

8.354.1 Detailed Description

Definition at line 97 of file vlx-vipc.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vipc.h

8.355 vipc_waiter_t Struct Reference

8.355.1 Detailed Description

Definition at line 55 of file vlx-vipc.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vipc.h

8.356 virtio_bus Struct Reference

8.356.1 Detailed Description

Definition at line 149 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.357 virtio_dev Struct Reference

8.357.1 Detailed Description

Definition at line 73 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.358 virtio_dev_info Struct Reference

8.358.1 Detailed Description

Definition at line 244 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.359 virtio_entry Union Reference

8.359.1 Detailed Description

Definition at line 47 of file vlx-virtio-bus.c.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.c

8.360 virtio_eventfd_match Struct Reference

8.360.1 Detailed Description

Definition at line 195 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.361 virtio_irqfd Struct Reference

8.361.1 Detailed Description

Definition at line 203 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.362 virtio_irqs Struct Reference

8.362.1 Detailed Description

Definition at line 64 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.363 virtio_memslot Struct Reference

8.363.1 Detailed Description

Definition at line 215 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.364 virtio_pmem Struct Reference

8.364.1 Detailed Description

Definition at line 47 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.365 virtio_regs Struct Reference

8.365.1 Detailed Description

Definition at line 59 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.366 virtio_ring Struct Reference

8.366.1 Detailed Description

Definition at line 140 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.367 virtio_rng Struct Reference

8.367.1 Detailed Description

Definition at line 47 of file virtio-rng.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/virtio-rng.c

8.368 virtio_vmem Struct Reference

8.368.1 Detailed Description

Definition at line 54 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.369 virtq_info Struct Reference

8.369.1 Detailed Description

Definition at line 266 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.370 virtq_info.__unnamed__ Union Reference

8.370.1 Detailed Description

Definition at line 283 of file vlx-virtio-bus.h.

The documentation for this union was generated from the following files:

8.371 virtq_info.__unnamed__.packed Struct Reference

8.371.1 Detailed Description

Definition at line 295 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following files:

8.372 virtq_info.__unnamed__.split Struct Reference

8.372.1 Detailed Description

Definition at line 284 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following files:

8.373 virtq_info_ops Struct Reference

8.373.1 Detailed Description

Definition at line 322 of file vlx-virtio-bus.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-virtio-bus.h

8.374 VL_ALIGN_64 Union Reference

8.374.1 Detailed Description

Definition at line 47 of file vdrv-types.h.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vgraphics/vdrv-types.h

8.375 VL_ALIGN_64.__unnamed__ Struct Reference

8.375.1 Detailed Description

Definition at line 49 of file vdrv-types.h.

The documentation for this struct was generated from the following files:

8.376 VlAtomic Struct Reference

8.376.1 Detailed Description

Definition at line 58 of file atomic.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/atomic.h

8.377 VlAtomic.__unnamed__ Union Reference

8.377.1 Detailed Description

Definition at line 59 of file atomic.h.

The documentation for this union was generated from the following files:

8.378 VIClientId Struct Reference

8.378.1 Detailed Description

Definition at line 33 of file vdrv-types.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vgraphics/vdrv-types.h

8.379 VIEvent Struct Reference

8.379.1 Detailed Description

Definition at line 37 of file event.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/event.h

8.380 vlib_param_lookup_ctxt_t Struct Reference

8.380.1 Detailed Description

Definition at line 24 of file vdriver-lib.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vdriver-lib.h

8.381 Vlink Struct Reference

8.381.1 Detailed Description

Definition at line 191 of file vlink-lib.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlink-lib.h

8.382 vlink2 Struct Reference

8.382.1 Detailed Description

Definition at line 236 of file vlink2.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlink2.h

8.383 vlink2_drv Struct Reference

8.383.1 Detailed Description

Definition at line 154 of file vlink2.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlink2.h

8.384 vlink2_op_desc Struct Reference

8.384.1 Detailed Description

Definition at line 193 of file vlink2.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlink2.h

8.385 vlink2_op_wrap Struct Reference

8.385.1 Detailed Description

Definition at line 198 of file vlink2.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlink2.h

8.386 vlink2_session Struct Reference

8.386.1 Detailed Description

Definition at line 280 of file vlink2.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlink2.h

8.387 vlink2_wait_queue Struct Reference

8.387.1 Detailed Description

Definition at line 30 of file vlink2.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlink2.h

8.388 vlink2_xirq_handle Struct Reference

8.388.1 Detailed Description

Definition at line 293 of file vlink2.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlink2.h

8.389 VlinkDrv Struct Reference

8.389.1 Detailed Description

Definition at line 110 of file vlink-lib.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlink-lib.h

8.390 VlinkOpDesc Struct Reference

8.390.1 Detailed Description

Definition at line 148 of file vlink-lib.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlink-lib.h

8.391 VlinkOpWrap Struct Reference

8.391.1 Detailed Description

Definition at line 153 of file vlink-lib.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlink-lib.h

8.392 VlinkSession Struct Reference

8.392.1 Detailed Description

Definition at line 234 of file vlink-lib.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlink-lib.h

8.393 VlinkXirqHandle Struct Reference

8.393.1 Detailed Description

Definition at line 247 of file vlink-lib.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlink-lib.h

8.394 VlListHead Struct Reference

8.394.1 Detailed Description

Definition at line 46 of file list.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/list.h

8.395 VlMemHead Struct Reference

8.395.1 Detailed Description

Definition at line 267 of file osal.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/osal.c

8.396 VlMemTail Struct Reference

8.396.1 Detailed Description

Definition at line 273 of file osal.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/osal.c

8.397 VlMutex Struct Reference

8.397.1 Detailed Description

Definition at line 34 of file mutex.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/mutex.h

8.398 VlOpaque Struct Reference

8.398.1 Detailed Description

Definition at line 27 of file base.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/base.h

8.399 VlSemaphore Struct Reference

8.399.1 Detailed Description

Definition at line 30 of file semaphore.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/semaphore.h

8.400 VlSpinlock Struct Reference

8.400.1 Detailed Description

Definition at line 28 of file spinlock.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/spinlock.h

8.401 VlThread Struct Reference

8.401.1 Detailed Description

Definition at line 26 of file thread.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/osal/thread.h

8.402 vlx_clk_req_t Struct Reference

Data Structures

- union u

8.402.1 Detailed Description

Definition at line 55 of file vlx-clk-ctrl-common.h.

8.402.2 Data Structure Documentation

8.402.2.1 union vlx_clk_req_t::u

Definition at line 62 of file vlx-clk-ctrl-common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-clk-ctrl-common.h

8.403 vlx_clk_res_t Struct Reference

8.403.1 Detailed Description

Definition at line 69 of file vlx-clk-ctrl-common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-clk-ctrl-common.h

8.404 vlx_dt Struct Reference

8.404.1 Detailed Description

Definition at line 30 of file vlx-dt.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-dt.h

8.405 vlx_dt_driver Struct Reference

8.405.1 Detailed Description

Definition at line 35 of file vlx-dt-drv.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-dt-drv.c

8.406 vlx_dt_vprop Struct Reference

8.406.1 Detailed Description

Definition at line 34 of file vlx-dt-vprop.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-dt-vprop.h

8.407 vlx_dt_vprop.__unnamed__ Union Reference

8.407.1 Detailed Description

Definition at line 40 of file vlx-dt-vprop.h.

The documentation for this union was generated from the following files:

8.408 vlx_evtlog_filtering_t Struct Reference

8.408.1 Detailed Description

Definition at line 54 of file vlx-event-log-filtering.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-event-log-filtering.c

8.409 vlx_history_t Struct Reference

8.409.1 Detailed Description

Definition at line 67 of file vlx-history.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-history.c

8.410 vlx_mhp Struct Reference

represent a device-tree node compatible with vl,vm-memory-hotplug

8.410.1 Detailed Description

represent a device-tree node compatible with vl,vm-memory-hotplug

Definition at line 74 of file vlx-memory-hotplug.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-memory-hotplug.c

8.411 vlx_mhp_list Struct Reference

global structure, referencing all struct vlx_mhp objects

Data Structures

- struct vlx_mhp_sysfs

8.411.1 Detailed Description

global structure, referencing all struct vlx_mhp objects

Definition at line 92 of file vlx-memory-hotplug.c.

8.411.2 Data Structure Documentation

8.411.2.1 struct vlx_mhp_list::vlx_mhp_sysfs

Definition at line 98 of file vlx-memory-hotplug.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-memory-hotplug.c

8.412 vlx_mhp_resource Struct Reference

8.412.1 Detailed Description

Definition at line 53 of file vlx-memory-hotplug.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-memory-hotplug.c

8.413 vlx_pd_req_t Struct Reference

Data Structures

- union u

8.413.1 Detailed Description

Definition at line 58 of file vlx-pm-domain.h.

8.413.2 Data Structure Documentation

8.413.2.1 union vlx_pd_req_t::u

Definition at line 62 of file vlx-pm-domain.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain.h

8.414 vlx_pd_res_t Struct Reference

8.414.1 Detailed Description

Definition at line 69 of file vlx-pm-domain.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-pm-domain.h

8.415 vlx_prop_t Struct Reference

8.415.1 Detailed Description

Definition at line 46 of file vlx-prop.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-prop.c

8.416 vlx_virtio_bind_eventfd Struct Reference

8.416.1 Detailed Description

Definition at line 67 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.417 vlx_virtio_dev_cb Struct Reference

VirtIO device-specific operations.

```
#include <vlx-virtio.h>
```

Data Fields

- void(∗ start)(struct vlx_virtio_dev ∗dev)

  Invoked by the framework when the front-end VM starts.
- void(∗ stop)(struct vlx_virtio_dev ∗dev)

  Invoked by the framework when the front-end VM stops.
- int(∗ queue_ready)(struct vlx_virtio_dev ∗dev, struct vlx_virtq ∗vq)

  Invoked by the framework when the front-end driver has finished configuring one of the virtqueue of this device and is ready to use it.
- void(∗ queue_idle)(struct vlx_virtio_dev ∗dev, struct vlx_virtq ∗vq)

  Invoked by the framework when the front-end driver wants to stop using a queue of this device.
- void(∗ status_changed)(struct vlx_virtio_dev ∗dev)

  Invoked by the framework when the driver has changed the device status bits.
- void(∗ apply_features)(struct vlx_virtio_dev ∗dev, u64 features)

  Invoked by the framework when the driver has negociated new features.
- u64(∗ load_config)(struct vlx_virtio_dev ∗dev, u32 offset, u32 count)

  Invoked by the framework when the driver loads a value from the device-specific configuration space.
- void(∗ store_config)(struct vlx_virtio_dev ∗dev, u32 offset, u32 count, u64 data)

  Invoked by the framework when the driver stores a value to the device-specific configuration space.

8.417.1 Detailed Description

VirtIO device-specific operations.

The VLX VirtIO framework handles common operations on VirtIO devices. For device-specific operations the framework calls into the back-end driver though the interface defined here.

All callbacks defined here are invoked sequentially using a worker thread on the VirtIO device work queue.

Definition at line 175 of file vlx-virtio.h.

8.417.2 Field Documentation

8.417.2.1 start

```
void(∗ vlx_virtio_dev_cb::start) (struct vlx_virtio_dev ∗dev)
```

Invoked by the framework when the front-end VM starts.

Parameters

in	*dev*	Reference to the VirtIO device.

Definition at line 181 of file vlx-virtio.h.

8.417.2.2 stop

```
void(* vlx_virtio_dev_cb::stop) (struct vlx_virtio_dev *dev)
```

Invoked by the framework when the front-end VM stops.

Parameters

in	*dev*	Reference to the VirtIO device.

Definition at line 188 of file vlx-virtio.h.

8.417.2.3 queue_ready

```
int(* vlx_virtio_dev_cb::queue_ready) (struct vlx_virtio_dev *dev, struct vlx_virtq *vq)
```

Invoked by the framework when the front-end driver has finished configuring one of the virtqueue of this device and is ready to use it.

Parameters

in	*dev*	Reference to the VirtIO device.
in	*vq*	Reference to the virtqueue that was setup by the front-end drivers.

Returns

0 if the back-end driver is ready to serve requests on this queue, a negative error code otherwise.

Definition at line 201 of file vlx-virtio.h.

8.417.2.4 queue_idle

```
void(* vlx_virtio_dev_cb::queue_idle) (struct vlx_virtio_dev *dev, struct vlx_virtq *vq)
```

Invoked by the framework when the front-end driver wants to stop using a queue of this device.

Parameters

in	*dev*	Reference to the VirtIO device.
in	*vq*	Reference to the virtqueue that the driver is no longer using.

Starting with VirtIO 1.0 specification, a driver can stop using one of the virtqueue of the device by writing 0 to the `QueueReady` register.

Definition at line 215 of file vlx-virtio.h.

8.417.2.5 status_changed

`void(* vlx_virtio_dev_cb::status_changed) (struct vlx_virtio_dev *dev)`

Invoked by the framework when the driver has changed the device status bits.

Parameters

in	*dev*	Reference to the VirtIO device.

Definition at line 223 of file vlx-virtio.h.

8.417.2.6 apply_features

`void(* vlx_virtio_dev_cb::apply_features) (struct vlx_virtio_dev *dev, u64 features)`

Invoked by the framework when the driver has negociated new features.

Parameters

in	*dev*	Reference to the VirtIO device.
in	*features*	The feature bits requested by the driver.

Definition at line 232 of file vlx-virtio.h.

8.417.2.7 load_config

`u64(* vlx_virtio_dev_cb::load_config) (struct vlx_virtio_dev *dev, u32 offset, u32 count)`

Invoked by the framework when the driver loads a value from the device-specific configuration space.

Parameters

in	*dev*	Reference to the VirtIO device.
in	*offset*	Offset in bytes within the device-specific configuration space.
in	*count*	Number of bytes to be loaded. `count` value will always be either 1, 2, 4 or 8.

Definition at line 244 of file vlx-virtio.h.

8.417.2.8 store_config

```
void(* vlx_virtio_dev_cb::store_config) (struct vlx_virtio_dev *dev, u32 offset, u32 count, u64
data)
```

Invoked by the framework when the driver stores a value to the device-specific configuration space.

Parameters

in	*dev*	Reference to the VirtIO device.
in	*offset*	Offset in bytes within the device-specific configuration space.
in	*count*	Number of bytes to be stored. `count` value will always be either 1, 2, 4 or 8.
in	*data*	The value to write to the configuration space.

Definition at line 257 of file vlx-virtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-virtio.h

8.418 vlx_virtio_dev_irqs Struct Reference

8.418.1 Detailed Description

Definition at line 62 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.419 vlx_virtio_dev_reg Struct Reference

8.419.1 Detailed Description

Definition at line 51 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.420 vlx_virtio_dev_regs Struct Reference

8.420.1 Detailed Description

Definition at line 56 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.421 vlx_virtio_event Struct Reference

8.421.1 Detailed Description

Definition at line 22 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.422 vlx_virtio_irqfd Struct Reference

8.422.1 Detailed Description

Definition at line 82 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.423 vlx_virtio_load_ack Struct Reference

8.423.1 Detailed Description

Definition at line 45 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.424 vlx_virtio_mapping Struct Reference

8.424.1 Detailed Description

Definition at line 91 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.425 vlx_virtio_post_irq Struct Reference

8.425.1 Detailed Description

Definition at line 74 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.426 vlx_virtio_vm_memory Struct Reference

8.426.1 Detailed Description

Definition at line 39 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.427 vlx_virtio_vm_region Struct Reference

8.427.1 Detailed Description

Definition at line 32 of file vlx-uvirtio.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-uvirtio.h

8.428 vlx_virtq Struct Reference

Representation of a virtqueue.

```
#include <vlx-virtq.h>
```

Data Fields

- struct vlx_virtio_dev ∗ dev
- unsigned int num
- unsigned int qsize
- unsigned int max_qsize
- enum kick_policy kick_policy
- void(∗ kick)(struct vlx_virtq ∗vq)

  Kick notification callbacks for this queue.
- void ∗ private_data

8.428.1 Detailed Description

Representation of a virtqueue.

The back-end driver is responsible of allocating the vlx_virtq objects during initialization before registering the device for each virtqueue that might be supported by the device.

Note that the front-end driver is still responsible for configuring the virtqueue and might not use all the virtqueues provided by the device.

Definition at line 75 of file vlx-virtq.h.

8.428.2 Field Documentation

8.428.2.1 dev

`struct vlx_virtio_dev* vlx_virtq::dev`

Pointer to the device this queue belongs to.

Definition at line 77 of file vlx-virtq.h.

8.428.2.2 num

`unsigned int vlx_virtq::num`

The queue number within the device.

Definition at line 80 of file vlx-virtq.h.

8.428.2.3 qsize

`unsigned int vlx_virtq::qsize`

Actual queue size configured by the driver.

Definition at line 83 of file vlx-virtq.h.

8.428.2.4 max_qsize

```
unsigned int vlx_virtq::max_qsize
```

Maximum queue size configured by the device.

Definition at line 86 of file vlx-virtq.h.

8.428.2.5 kick_policy

```
enum kick_policy vlx_virtq::kick_policy
```

Specifies how kick notification callbacks will be invoked. Please refer to kick_policy documentation for the different options provided by the VLX VirtIO framework.

Definition at line 93 of file vlx-virtq.h.

8.428.2.6 kick

```
void(* vlx_virtq::kick) (struct vlx_virtq *vq)
```

Kick notification callbacks for this queue.

Parameters

in	*vq*	Reference to the virtqueue.

Definition at line 100 of file vlx-virtq.h.

8.428.2.7 private_data

```
void* vlx_virtq::private_data
```

Back-end driver private data.

Definition at line 103 of file vlx-virtq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-virtq.h

8.429 vlx_vm_cpu_hotplug_state_t Struct Reference

8.429.1 Detailed Description

Definition at line 92 of file vlx-cpu-hotplug.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-cpu-hotplug.c

8.430 vlx_vm_vcpu_cpu_id_info_t Struct Reference

8.430.1 Detailed Description

Definition at line 84 of file vlx-cpu-hotplug.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-cpu-hotplug.c

8.431 vmbox Struct Reference

8.431.1 Detailed Description

Definition at line 66 of file vmbox.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vmbox.c

8.432 vmbox_link Struct Reference

8.432.1 Detailed Description

Definition at line 52 of file vmbox.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vmbox.c

8.433 vmbox_test_ctx Struct Reference

8.433.1 Detailed Description

Definition at line 73 of file vmbox-test.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vmbox-test.c

8.434 vmbox_test_msg Struct Reference

8.434.1 Detailed Description

Definition at line 55 of file vmbox-test.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vmbox-test.c

8.435 VMQ_ALIGN Struct Reference

8.435.1 Detailed Description

Definition at line 130 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.436 vmq_callbacks_t Struct Reference

8.436.1 Detailed Description

Definition at line 80 of file vlx-vmq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.h

8.437 vmq_desc Struct Reference

8.437.1 Detailed Description

Definition at line 150 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.438 vmq_is Struct Reference

8.438.1 Detailed Description

Definition at line 142 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.439 vmq_link_public_t Struct Reference

8.439.1 Detailed Description

Definition at line 35 of file vlx-vmq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.h

8.440 vmq_link_t Struct Reference

8.440.1 Detailed Description

Definition at line 1513 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.441 vmq_links_public_t Struct Reference

8.441.1 Detailed Description

Definition at line 57 of file vlx-vmq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.h

8.442 vmq_links_t Struct Reference

8.442.1 Detailed Description

Definition at line 1669 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.443 vmq_rx Struct Reference

8.443.1 Detailed Description

Definition at line 1411 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.444 vmq_ss Struct Reference

8.444.1 Detailed Description

Definition at line 283 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.445 vmq_tx Struct Reference

Data Structures

- struct vmq_tx_errors
- struct vmq_tx_warnings

8.445.1 Detailed Description

Definition at line 831 of file vlx-vmq.c.

8.445.2 Data Structure Documentation

8.445.2.1 struct vmq_tx::vmq_tx_errors

Definition at line 869 of file vlx-vmq.c.

8.445.2.2 struct vmq_tx::vmq_tx_warnings

Definition at line 850 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.446 vmq_tx.stats Struct Reference

8.446.1 Detailed Description

Definition at line 838 of file vlx-vmq.c.

The documentation for this struct was generated from the following files:

8.447 vmq_xx Struct Reference

Data Structures

- struct vmq_xx_errors
- struct vmq_xx_warnings

8.447.1 Detailed Description

Definition at line 235 of file vlx-vmq.c.

8.447.2 Data Structure Documentation

8.447.2.1 struct vmq_xx::vmq_xx_errors

Definition at line 275 of file vlx-vmq.c.

8.447.2.2 struct vmq_xx::vmq_xx_warnings

Definition at line 271 of file vlx-vmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.c

8.448 vmq_xx_config_t Struct Reference

8.448.1 Detailed Description

Definition at line 70 of file vlx-vmq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vmq.h

8.449 vmqt_link_t Struct Reference

8.449.1 Detailed Description

Definition at line 81 of file vmq-tests.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vmq-tests.c

8.450 vmqt_req_t Struct Reference

8.450.1 Detailed Description

Definition at line 77 of file vmq-tests.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vmq-tests.c

8.451 VRing Struct Reference

8.451.1 Detailed Description

Definition at line 35 of file vevdev_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vevdev_common.h

8.452 vrpc_end Struct Reference

8.452.1 Detailed Description

Definition at line 78 of file vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpc.c

8.453 vrpc_ep_t Struct Reference

8.453.1 Detailed Description

Definition at line 62 of file vrpc-test.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpc-test.c

8.454 vrpc_pmem_t Struct Reference

8.454.1 Detailed Description

Definition at line 29 of file vrpc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpc_common.h

8.455 vrpc_t Struct Reference

8.455.1 Detailed Description

Definition at line 87 of file vrpc.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpc.c

8.456 vrpc_thread_ops Struct Reference

8.456.1 Detailed Description

Definition at line 173 of file vrpc.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpc.h

8.457 VrpqAdmFlags Struct Reference

8.457.1 Detailed Description

Definition at line 515 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.458 VrpqAdmFlags.__unnamed__ Union Reference

8.458.1 Detailed Description

Definition at line 516 of file vrpq.h.

The documentation for this union was generated from the following files:

8.459 VrpqAdmFlags.__unnamed__.__unnamed__ Struct Reference

8.459.1 Detailed Description

Definition at line 518 of file vrpq.h.

The documentation for this struct was generated from the following files:

8.460 VrpqBitmapAlloc Struct Reference

8.460.1 Detailed Description

Definition at line 502 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.461 VrpqCallback Struct Reference

8.461.1 Detailed Description

Definition at line 279 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.462 VrpqCallbackFlags Struct Reference

8.462.1 Detailed Description

Definition at line 343 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.463 VrpqCallbackFlags.__unnamed__ Union Reference

8.463.1 Detailed Description

Definition at line 344 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.464 VrpqCallbackFlags.__unnamed__.__unnamed__ Struct Reference

8.464.1 Detailed Description

Definition at line 346 of file vrpq-proto.h.

The documentation for this struct was generated from the following files:

8.465 VrpqCallbackInfo Struct Reference

8.465.1 Detailed Description

Definition at line 272 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/common.h

8.466 VrpqCallbackInfo.in Struct Reference

8.466.1 Detailed Description

Definition at line 274 of file common.h.

The documentation for this struct was generated from the following files:

8.467 VrpqCallbackInfo.out Union Reference

8.467.1 Detailed Description

Definition at line 282 of file common.h.

The documentation for this union was generated from the following files:

8.468 VrpqCallbackSlot Struct Reference

8.468.1 Detailed Description

Definition at line 380 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.469 VrpqCallbackSlot.__unnamed__ Union Reference

8.469.1 Detailed Description

Definition at line 385 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.470 VrpqChanAcceptOp Struct Reference

8.470.1 Detailed Description

Definition at line 155 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/ioctl.h

8.471 VrpqChanCreateIn Struct Reference

8.471.1 Detailed Description

Definition at line 530 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.472 VrpqChanCreateOp Struct Reference

8.472.1 Detailed Description

Definition at line 149 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/ioctl.h

8.473 VrpqChanCreateOut Struct Reference

8.473.1 Detailed Description

Definition at line 536 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.474 VrpqChanDestroyIn Struct Reference

8.474.1 Detailed Description

Definition at line 540 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.475 VrpqChanFlags Struct Reference

8.475.1 Detailed Description

Definition at line 106 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.476 VrpqChanFlags.__unnamed__ Union Reference

8.476.1 Detailed Description

Definition at line 107 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.477 VrpqChanFlags.__unnamed__.__unnamed__ Struct Reference

8.477.1 Detailed Description

Definition at line 109 of file vrpq-proto.h.

The documentation for this struct was generated from the following files:

8.478 VrpqChanReqRing Struct Reference

8.478.1 Detailed Description

Definition at line 489 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.479 VrpqChanStuckIn Struct Reference

8.479.1 Detailed Description

Definition at line 544 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.480 VrpqChunkFlags Struct Reference

8.480.1 Detailed Description

Definition at line 260 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.481 VrpqChunkFlags.__unnamed__ Union Reference

8.481.1 Detailed Description

Definition at line 261 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.482 VrpqChunkFlags.__unnamed__.__unnamed__ Struct Reference

8.482.1 Detailed Description

Definition at line 263 of file vrpq-proto.h.

The documentation for this struct was generated from the following files:

8.483 VrpqCloseFlags Struct Reference

8.483.1 Detailed Description

Definition at line 550 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.484 VrpqCloseFlags.__unnamed__ Union Reference

8.484.1 Detailed Description

Definition at line 551 of file vrpq.h.

The documentation for this union was generated from the following files:

8.485 VrpqCloseFlags.__unnamed__.__unnamed__ Struct Reference

8.485.1 Detailed Description

Definition at line 553 of file vrpq.h.

The documentation for this struct was generated from the following files:

8.486 VrpqCltCall Struct Reference

8.486.1 Detailed Description

Definition at line 336 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.487 VrpqCltCallAlloc Struct Reference

8.487.1 Detailed Description

Definition at line 349 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.488 VrpqCltCallIntr Struct Reference

8.488.1 Detailed Description

Definition at line 361 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.489 VrpqCltChan Struct Reference

8.489.1 Detailed Description

Definition at line 390 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.490 VrpqCltDev Struct Reference

8.490.1 Detailed Description

Definition at line 371 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.491 VrpqCltFile Struct Reference

8.491.1 Detailed Description

Definition at line 416 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.492 VrpqCltFile.__unnamed__ Union Reference

8.492.1 Detailed Description

Definition at line 420 of file vrpq.h.

The documentation for this union was generated from the following files:

8.493 VrpqCltIFOps Struct Reference

8.493.1 Detailed Description

Definition at line 461 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.494 VrpqCltSession Struct Reference

8.494.1 Detailed Description

Definition at line 404 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.495 VrpqCopyIn Struct Reference

8.495.1 Detailed Description

Definition at line 98 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.496 VrpqCopyIn.in Struct Reference

8.496.1 Detailed Description

Definition at line 99 of file vrpq-os.h.

The documentation for this struct was generated from the following files:

8.497 VrpqCopyOut Struct Reference

8.497.1 Detailed Description

Definition at line 106 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.498 VrpqCopyOut.out Struct Reference

8.498.1 Detailed Description

Definition at line 107 of file vrpq-os.h.

The documentation for this struct was generated from the following files:

8.499 VrpqDev Struct Reference

8.499.1 Detailed Description

Definition at line 789 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.500 VrpqDev.__unnamed__ Union Reference

8.500.1 Detailed Description

Definition at line 790 of file vrpq.h.

The documentation for this union was generated from the following files:

8.501 VrpqDevOps Struct Reference

8.501.1 Detailed Description

Definition at line 33 of file vrpq-drv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/vrpq-drv.h

8.502 VrpqDrv Struct Reference

8.502.1 Detailed Description

Definition at line 50 of file vrpq-drv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/vrpq-drv.h

8.503 VrpqFile Struct Reference

8.503.1 Detailed Description

Definition at line 804 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.504 VrpqFile.__unnamed__ Union Reference

8.504.1 Detailed Description

Definition at line 805 of file vrpq.h.

The documentation for this union was generated from the following files:

8.505 VrpqGenDev Struct Reference

8.505.1 Detailed Description

Definition at line 101 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.506 VrpqGenFile Struct Reference

8.506.1 Detailed Description

Definition at line 139 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.507 VrpqGlbStats Struct Reference

8.507.1 Detailed Description

Definition at line 196 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.508 VrpqGlbStats.__unnamed__ Union Reference

8.508.1 Detailed Description

Definition at line 197 of file vrpq.h.

The documentation for this union was generated from the following files:

8.509 VrpqGlbStats.__unnamed__.__unnamed__ Struct Reference

8.509.1 Detailed Description

Definition at line 198 of file vrpq.h.

The documentation for this struct was generated from the following files:

8.510 VrpqIFDev Struct Reference

8.510.1 Detailed Description

Definition at line 46 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.511 VrpqIFDrv Struct Reference

8.511.1 Detailed Description

Definition at line 41 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.512 VrpqIFFile Struct Reference

8.512.1 Detailed Description

Definition at line 50 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.513 VrpqIFOps Struct Reference

8.513.1 Detailed Description

Definition at line 797 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.514 VrpqIFOps.__unnamed__ Union Reference

8.514.1 Detailed Description

Definition at line 798 of file vrpq.h.

The documentation for this union was generated from the following files:

8.515 VrpqIFShm Union Reference

8.515.1 Detailed Description

Definition at line 73 of file vrpq-os.h.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.516 VrpqIFShm.clt Struct Reference

8.516.1 Detailed Description

Definition at line 77 of file vrpq-os.h.

The documentation for this struct was generated from the following files:

8.517 VrpqIFShm.srv Struct Reference

8.517.1 Detailed Description

Definition at line 74 of file vrpq-os.h.

The documentation for this struct was generated from the following files:

8.518 VrpqInfo Struct Reference

8.518.1 Detailed Description

Definition at line 252 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/common.h

8.519 VrpqIov Struct Reference

8.519.1 Detailed Description

Definition at line 86 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.520 Vrpqlov.__unnamed__ Union Reference

8.520.1 Detailed Description

Definition at line 87 of file vrpq-os.h.

The documentation for this union was generated from the following files:

8.521 Vrpqlov.__unnamed__ Union Reference

8.521.1 Detailed Description

Definition at line 87 of file vrpq-os.h.

The documentation for this union was generated from the following files:

8.522 VrpqMsgFlags Struct Reference

8.522.1 Detailed Description

Definition at line 125 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.523 VrpqMsgFlags.__unnamed__ Union Reference

8.523.1 Detailed Description

Definition at line 126 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.524 VrpqMsgFlags.__unnamed__.__unnamed__ Struct Reference

8.524.1 Detailed Description

Definition at line 128 of file vrpq-proto.h.

The documentation for this struct was generated from the following files:

8.525 VrpqParamAlloc Struct Reference

8.525.1 Detailed Description

Definition at line 320 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.526 VrpqParamCallInfo Struct Reference

8.526.1 Detailed Description

Definition at line 141 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/common.h

8.527 VrpqParamChunk Struct Reference

8.527.1 Detailed Description

Definition at line 273 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.528 VrpqParamCopy Struct Reference

8.528.1 Detailed Description

Definition at line 2878 of file vrpq-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-fe.c

8.529 VrpqParamCopy.__unnamed__ Union Reference

8.529.1 Detailed Description

Definition at line 2883 of file vrpq-fe.c.

The documentation for this union was generated from the following files:

8.530 VrpqParamCopy.__unnamed__ Union Reference

8.530.1 Detailed Description

Definition at line 2883 of file vrpq-fe.c.

The documentation for this union was generated from the following files:

8.531 VrpqParamInfo Struct Reference

8.531.1 Detailed Description

Definition at line 151 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/common.h

8.532 VrpqParamInfo.__unnamed__ Union Reference

8.532.1 Detailed Description

Definition at line 152 of file common.h.

The documentation for this union was generated from the following files:

8.533 VrpqParamPostInfo Struct Reference

8.533.1 Detailed Description

Definition at line 134 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/common.h

8.534 VrpqParamRef Struct Reference

8.534.1 Detailed Description

Definition at line 116 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/common.h

8.535 VrpqParamRef.__unnamed__ Union Reference

8.535.1 Detailed Description

Definition at line 121 of file common.h.

The documentation for this union was generated from the following files:

8.536 VrpqParamRef.__unnamed__ Union Reference

8.536.1 Detailed Description

Definition at line 121 of file common.h.

The documentation for this union was generated from the following files:

8.537 VrpqParamRing Struct Reference

8.537.1 Detailed Description

Definition at line 282 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.538 VrpqParamsShm Struct Reference

8.538.1 Detailed Description

Definition at line 54 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.539 VrpqParentContext Struct Reference

8.539.1 Detailed Description

Definition at line 19 of file vrpq-drv-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/vrpq-drv-os.h

8.540 VrpqPmemLayout Struct Reference

8.540.1 Detailed Description

Definition at line 458 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.541 VrpqPmemShm Struct Reference

8.541.1 Detailed Description

Definition at line 66 of file vrpq-os.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-os.h

8.542 VrpqProcInfo Struct Reference

8.542.1 Detailed Description

Definition at line 128 of file client-drv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/client-drv.h

8.543 VrpqProcReq Struct Reference

8.543.1 Detailed Description

Definition at line 85 of file server-drv.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/server-drv.h

8.544 VrpqProcStatEnt Struct Reference

8.544.1 Detailed Description

Definition at line 185 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.545 VrpqProcStatEnt.__unnamed__ Union Reference

8.545.1 Detailed Description

Definition at line 186 of file vrpq.h.

The documentation for this union was generated from the following files:

8.546 VrpqProcStatTbl Struct Reference

8.546.1 Detailed Description

Definition at line 192 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.547 VrpqProfOp Struct Reference

8.547.1 Detailed Description

Definition at line 161 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/ioctl.h

8.548 VrpqProtoInfo Struct Reference

8.548.1 Detailed Description

Definition at line 416 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.549 VrpqProtoInfo.__unnamed__ Union Reference

8.549.1 Detailed Description

Definition at line 417 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.550 VrpqProtoInfo.__unnamed__.__unnamed__ Struct Reference

8.550.1 Detailed Description

Definition at line 418 of file vrpq-proto.h.

The documentation for this struct was generated from the following files:

8.551 VrpqReqAlloc Struct Reference

8.551.1 Detailed Description

Definition at line 313 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.552 VrpqReqMgr Struct Reference

8.552.1 Detailed Description

Definition at line 325 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.553 VrpqReqMsg Struct Reference

8.553.1 Detailed Description

Definition at line 156 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.554 VrpqReqMsg.__unnamed__ Union Reference

8.554.1 Detailed Description

Definition at line 162 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.555 VrpqReqPeer Struct Reference

8.555.1 Detailed Description

Definition at line 307 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.556 VrpqReqRing Struct Reference

8.556.1 Detailed Description

Definition at line 197 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.557 VrpqReqRingGbl Struct Reference

8.557.1 Detailed Description

Definition at line 186 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.558 VrpqRingFlags Struct Reference

8.558.1 Detailed Description

Definition at line 173 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.559 VrpqRingFlags.__unnamed__ Union Reference

8.559.1 Detailed Description

Definition at line 174 of file vrpq-proto.h.

The documentation for this union was generated from the following files:

8.560 VrpqRingFlags.__unnamed__.__unnamed__ Struct Reference

8.560.1 Detailed Description

Definition at line 176 of file vrpq-proto.h.

The documentation for this struct was generated from the following files:

8.561 VrpqRspAlloc Struct Reference

8.561.1 Detailed Description

Definition at line 484 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.562 VrpqRspMsg Struct Reference

8.562.1 Detailed Description

Definition at line 223 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.563 VrpqRspPeer Struct Reference

8.563.1 Detailed Description

Definition at line 508 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.564 VrpqRspRing Struct Reference

8.564.1 Detailed Description

Definition at line 240 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.565 VrpqRspRingGbl Struct Reference

8.565.1 Detailed Description

Definition at line 231 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.566 VrpqSessionAcceptOp Struct Reference

8.566.1 Detailed Description

Definition at line 142 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/ioctl.h

8.567 VrpqSessionCreateIn Struct Reference

8.567.1 Detailed Description

Definition at line 515 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.568 VrpqSessionCreateOp Struct Reference

8.568.1 Detailed Description

Definition at line 135 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/ioctl.h

8.569 VrpqSessionCreateOut Struct Reference

8.569.1 Detailed Description

Definition at line 522 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.570 VrpqSessionDestroyIn Struct Reference

8.570.1 Detailed Description

Definition at line 526 of file vrpq-proto.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq-proto.h

8.571 VrpqShmMap Struct Reference

8.571.1 Detailed Description

Definition at line 149 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.572 VrpqShmMapOp Struct Reference

8.572.1 Detailed Description

Definition at line 167 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/ioctl.h

8.573 VrpqShmSetupOp Struct Reference

8.573.1 Detailed Description

Definition at line 173 of file ioctl.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/ioctl.h

8.574 VrpqSrvAdmReq Struct Reference

8.574.1 Detailed Description

Definition at line 575 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.575 VrpqSrvAdmWaiter Struct Reference

8.575.1 Detailed Description

Definition at line 582 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.576 VrpqSrvChan Struct Reference

8.576.1 Detailed Description

Definition at line 560 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.577 VrpqSrvDev Struct Reference

8.577.1 Detailed Description

Definition at line 528 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.578 VrpqSrvFile Struct Reference

8.578.1 Detailed Description

Definition at line 607 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.579 VrpqSrvFile.__unnamed__ Union Reference

8.579.1 Detailed Description

Definition at line 611 of file vrpq.h.

The documentation for this union was generated from the following files:

8.580 VrpqSrvIFOps Struct Reference

8.580.1 Detailed Description

Definition at line 648 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.581 VrpqSrvSession Struct Reference

8.581.1 Detailed Description

Definition at line 593 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.582 VrpqSrvSessionAdm Struct Reference

8.582.1 Detailed Description

Definition at line 587 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.583 VrpqStat Struct Reference

8.583.1 Detailed Description

Definition at line 373 of file common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrpq/common.h

8.584 VrpqStats Struct Reference

8.584.1 Detailed Description

Definition at line 258 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.585 VrpqXirqMap Struct Reference

8.585.1 Detailed Description

Definition at line 90 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.586 vrtc_ipc_t Union Reference

8.586.1 Detailed Description

Definition at line 80 of file vrtc_common.h.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.587 vrtc_req_t Struct Reference

8.587.1 Detailed Description

Definition at line 49 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.588 vrtc_req_time_t Struct Reference

8.588.1 Detailed Description

Definition at line 54 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.589 vrtc_req_wkalrm_t Struct Reference

8.589.1 Detailed Description

Definition at line 59 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.590 vrtc_res_t Struct Reference

8.590.1 Detailed Description

Definition at line 65 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.591 vrtc_res_time_t Struct Reference

8.591.1 Detailed Description

Definition at line 70 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.592 vrtc_res_wkalrm_t Struct Reference

8.592.1 Detailed Description

Definition at line 75 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.593 vrtc_t Struct Reference

8.593.1 Detailed Description

Definition at line 77 of file vrtc-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrtc-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrtc-fe.c

8.594 vrtc_time_t Struct Reference

8.594.1 Detailed Description

Definition at line 30 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.595 vrtc_wkalrm_t Struct Reference

8.595.1 Detailed Description

Definition at line 42 of file vrtc_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vrtc_common.h

8.596 vsig_ctl Struct Reference

8.596.1 Detailed Description

Definition at line 185 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.597 vsig_mask Struct Reference

8.597.1 Detailed Description

Definition at line 180 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.598 vsig_peer Struct Reference

8.598.1 Detailed Description

Definition at line 212 of file vfence2.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2.c

8.599 VsigCtl Struct Reference

8.599.1 Detailed Description

Definition at line 27 of file vsig.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vsig.h

8.600 VsigMask Struct Reference

8.600.1 Detailed Description

Definition at line 22 of file vsig.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vsig.h

8.601 VsigMgr Struct Reference

8.601.1 Detailed Description

Definition at line 273 of file vrpq.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vrpq.h

8.602 VsigPeer Struct Reference

8.602.1 Detailed Description

Definition at line 17 of file vsig.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vsig.h

8.603 vsmq Struct Reference

8.603.1 Detailed Description

Definition at line 82 of file vsmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vsmq.c

8.604 vsmq.index Union Reference

8.604.1 Detailed Description

Definition at line 104 of file vsmq.c.

The documentation for this union was generated from the following files:

8.605 vsmq_count Struct Reference

8.605.1 Detailed Description

Definition at line 65 of file vsmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vsmq.c

8.606 vsmq_head Struct Reference

8.606.1 Detailed Description

Definition at line 60 of file vsmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vsmq.c

8.607 vsmq_proc Struct Reference

8.607.1 Detailed Description

Definition at line 77 of file vsmq.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vsmq.c

8.608 vsmqt_driver Struct Reference

8.608.1 Detailed Description

Definition at line 63 of file vsmq-test.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vsmq-test.c

8.609 vstress_drv_t Struct Reference

8.609.1 Detailed Description

Definition at line 164 of file vlx-vstress.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vstress.c

8.610 vstress_drv_t.stats Struct Reference

8.610.1 Detailed Description

Definition at line 193 of file vlx-vstress.c.

The documentation for this struct was generated from the following files:

8.611 vstress_drv_t.test Struct Reference

8.611.1 Detailed Description

Definition at line 183 of file vlx-vstress.c.

The documentation for this struct was generated from the following files:

8.612 vstress_drv_t.warnings Struct Reference

8.612.1 Detailed Description

Definition at line 188 of file vlx-vstress.c.

The documentation for this struct was generated from the following files:

8.613 vstress_link_t Struct Reference

8.613.1 Detailed Description

Definition at line 131 of file vlx-vstress.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vstress.c

8.614 vstress_link_t.stats Struct Reference

8.614.1 Detailed Description

Definition at line 138 of file vlx-vstress.c.

The documentation for this struct was generated from the following files:

8.615 vstress_proc_t Struct Reference

8.615.1 Detailed Description

Definition at line 902 of file vlx-vstress.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vstress.c

8.616 vstress_thread_t Struct Reference

8.616.1 Detailed Description

Definition at line 144 of file vlx-vstress.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-vstress.c

8.617 vthermal Struct Reference

8.617.1 Detailed Description

Definition at line 28 of file vthermal-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vthermal-be.c

8.618 vthermal_data Struct Reference

8.618.1 Detailed Description

Definition at line 47 of file vthermal-common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vthermal-common.h

8.619 vthermal_fe Struct Reference

8.619.1 Detailed Description

Definition at line 30 of file vthermal-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vthermal-fe.c

8.620 VVideo2Abort Struct Reference

8.620.1 Detailed Description

Definition at line 74 of file vvideo2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vvideo2_common.h

8.621 VVideo2Ioctl Struct Reference

8.621.1 Detailed Description

Definition at line 57 of file vvideo2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vvideo2_common.h

8.622 VVideo2Ioctl.__unnamed__ Union Reference

8.622.1 Detailed Description

Definition at line 64 of file vvideo2_common.h.

The documentation for this union was generated from the following files:

8.623 VVideo2Ioctl.__unnamed__.__unnamed__ Struct Reference

8.623.1 Detailed Description

Definition at line 65 of file vvideo2_common.h.

The documentation for this struct was generated from the following files:

8.624 VVideo2Open Struct Reference

8.624.1 Detailed Description

Definition at line 53 of file vvideo2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vvideo2_common.h

8.625 VVideo2Poll Struct Reference

8.625.1 Detailed Description

Definition at line 79 of file vvideo2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vvideo2_common.h

8.626 VVideo2Request Struct Reference

8.626.1 Detailed Description

Definition at line 91 of file vvideo2_common.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vvideo2_common.h

8.627 VVideo2Request.u Union Reference

8.627.1 Detailed Description

Definition at line 100 of file vvideo2_common.h.

The documentation for this union was generated from the following files:

8.628 vvideo_be_t Struct Reference

8.628.1 Detailed Description

Definition at line 602 of file vvideo2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c

8.629 vvideo_be_t.ctx_put Struct Reference

8.629.1 Detailed Description

Definition at line 661 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.630 vvideo_be_t.stats Struct Reference

8.630.1 Detailed Description

Definition at line 677 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.631 vvideo_be_t.stats.kill_pid Struct Reference

8.631.1 Detailed Description

Definition at line 691 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.632 vvideo_be_t.warnings Struct Reference

8.632.1 Detailed Description

Definition at line 668 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.633 vvideo_call_t Struct Reference

8.633.1 Detailed Description

Definition at line 700 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.634 vvideo_config_t Struct Reference

8.634.1 Detailed Description

Definition at line 149 of file vvideo-util.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo-util.h

8.635 vvideo_configs_t Struct Reference

8.635.1 Detailed Description

Definition at line 156 of file vvideo-util.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo-util.h

8.636 vvideo_ctx_iter_t Struct Reference

8.636.1 Detailed Description

Definition at line 557 of file vvideo2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.637 vvideo_ctx_t Struct Reference

8.637.1 Detailed Description

Definition at line 340 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.638 vvideo_ctx_t.kref_put Struct Reference

8.638.1 Detailed Description

Definition at line 369 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.639 vvideo_ctx_t.opener Struct Reference

8.639.1 Detailed Description

Definition at line 651 of file vvideo2-fe.c.

The documentation for this struct was generated from the following files:

8.640 vvideo_ctx_t.poll Struct Reference

8.640.1 Detailed Description

Definition at line 363 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.641 vvideo_ctx_t.poll Struct Reference

8.641.1 Detailed Description

Definition at line 363 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.642 vvideo_dev_t Struct Reference

Data Structures

- struct vdev_errors
- struct vdev_errors.fd2gid
- struct vdev_errors.gid2fd
- struct vdev_stats
- struct vdev_stats.ioctl
- struct vdev_stats.poll
- struct vdev_warnings
- struct vdev_warnings.fd2gid
- struct vdev_warnings.gid2fd
- struct vdev_warnings.poll

8.642.1 Detailed Description

Definition at line 483 of file vvideo2-be.c.

8.642.2 Data Structure Documentation

8.642.2.1 struct vvideo_dev_t::vdev_errors

Definition at line 495 of file vvideo2-be.c.

8.642.2.2 struct vvideo_dev_t::vdev_errors.fd2gid

Definition at line 502 of file vvideo2-be.c.

8.642.2.3 struct vvideo_dev_t::vdev_errors.gid2fd

Definition at line 446 of file vvideo2-fe.c.

8.642.2.4 struct vvideo_dev_t::vdev_stats

Definition at line 557 of file vvideo2-be.c.

8.642.2.5 struct vvideo_dev_t::vdev_stats.ioctl

Definition at line 543 of file vvideo2-fe.c.

8.642.2.6 struct vvideo_dev_t::vdev_stats.poll

Definition at line 575 of file vvideo2-be.c.

8.642.2.7 struct vvideo_dev_t::vdev_warnings

Definition at line 526 of file vvideo2-be.c.

8.642.2.8 struct vvideo_dev_t::vdev_warnings.fd2gid

Definition at line 550 of file vvideo2-be.c.

8.642.2.9 struct vvideo_dev_t::vdev_warnings.gid2fd

Definition at line 488 of file vvideo2-fe.c.

8.642.2.10 struct vvideo_dev_t::vdev_warnings.poll

Definition at line 538 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.643 vvideo_dmabuf_gid_t Struct Reference

8.643.1 Detailed Description

Definition at line 343 of file vvideo-util.h.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo-util.h

8.644 vvideo_dmabuf_t Struct Reference

8.644.1 Detailed Description

Definition at line 269 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.645 vvideo_fe_t Struct Reference

8.645.1 Detailed Description

Definition at line 576 of file vvideo2-fe.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.646 vvideo_fe_t.stats Struct Reference

8.646.1 Detailed Description

Definition at line 624 of file vvideo2-fe.c.

The documentation for this struct was generated from the following files:

8.647 vvideo_fe_t.warnings Struct Reference

8.647.1 Detailed Description

Definition at line 617 of file vvideo2-fe.c.

The documentation for this struct was generated from the following files:

8.648 vvideo_link_t Struct Reference

8.648.1 Detailed Description

Definition at line 379 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.649 vvideo_link_t.events Struct Reference

8.649.1 Detailed Description

Definition at line 402 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.650 vvideo_link_t.secure Struct Reference

8.650.1 Detailed Description

Definition at line 419 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.651 vvideo_link_t.stats Struct Reference

8.651.1 Detailed Description

Definition at line 300 of file vvideo2-fe.c.

The documentation for this struct was generated from the following files:

8.652 vvideo_link_t.stats Struct Reference

8.652.1 Detailed Description

Definition at line 300 of file vvideo2-fe.c.

The documentation for this struct was generated from the following files:

8.653 vvideo_link_t.warnings Struct Reference

8.653.1 Detailed Description

Definition at line 306 of file vvideo2-fe.c.

The documentation for this struct was generated from the following files:

8.654 vvideo_msg_t Struct Reference

8.654.1 Detailed Description

Definition at line 438 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.655 vvideo_msg_t.ioctl Struct Reference

8.655.1 Detailed Description

Definition at line 364 of file vvideo2-fe.c.

The documentation for this struct was generated from the following files:

8.656 vvideo_poll_t Struct Reference

8.656.1 Detailed Description

Definition at line 315 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.657 vvideo_poll_table_entry_t Struct Reference

8.657.1 Detailed Description

Definition at line 306 of file vvideo2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c

8.658 vvideo_secure_t Union Reference

8.658.1 Detailed Description

Definition at line 432 of file vvideo2-be.c.

The documentation for this union was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c

8.659 vvideo_thread_t Struct Reference

8.659.1 Detailed Description

Definition at line 452 of file vvideo2-be.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c

8.660 vvideo_thread_t.ioctl Struct Reference

8.660.1 Detailed Description

Definition at line 471 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

8.661 vvideo_v4l2_exportbuffer Struct Reference

8.661.1 Detailed Description

Definition at line 706 of file vvideo2-be.c.

The documentation for this struct was generated from the following files:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-be.c
- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vvideo2-fe.c

8.662 vwatchdog_dev_t Struct Reference

8.662.1 Detailed Description

Definition at line 134 of file vwatchdog.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vwatchdog.c

8.663 vwatchdog_drv_t Struct Reference

8.663.1 Detailed Description

Definition at line 119 of file vwatchdog.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vwatchdog.c

8.664 vwatchdog_prop_t Struct Reference

8.664.1 Detailed Description

Definition at line 126 of file vwatchdog.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vwatchdog.c

8.665 xirqb_desc_t Struct Reference

8.665.1 Detailed Description

Definition at line 175 of file xirq-bench.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/xirq-bench.c

8.666 xirqb_distrib_interval_t Struct Reference

8.666.1 Detailed Description

Definition at line 154 of file xirq-bench.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/xirq-bench.c

8.667 xirqb_distrib_t Struct Reference

8.667.1 Detailed Description

Definition at line 139 of file xirq-bench.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/xirq-bench.c

8.668 xirqb_remote_t Struct Reference

8.668.1 Detailed Description

Definition at line 161 of file xirq-bench.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/xirq-bench.c

8.669 xirqb_sample_t Struct Reference

8.669.1 Detailed Description

Definition at line 131 of file xirq-bench.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/xirq-bench.c

8.670 xpic_chip_data Struct Reference

8.670.1 Detailed Description

Definition at line 58 of file vlx-xpic.c.

The documentation for this struct was generated from the following file:

- vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vlx-xpic.c

Chapter 9

File Documentation

9.1 linux-drivers-vl-3.18/linux-kernel-vdrivers/doc/01_admin_8.dox File Reference

9.2 linux-drivers-vl-3.18/linux-kernel-vdrivers/doc/01_files_5.dox File Reference

9.3 linux-drivers-vl-3.18/linux-kernel-vdrivers/doc/01_nkddi_3D.dox File Reference

9.4 linux-drivers-vl-3.18/linux-kernel-vdrivers/doc/01_vdrivers_4D.dox File Reference

9.5 linux-drivers-vl-3.18/linux-kernel-vdrivers/doc/manpages.dox File Reference

9.6 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/htf/htf-vgic-vdev.c File Reference

This file contains a HTF virtual device driver for testing vGIC emulation.

```
#include <linux/version.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/uaccess.h>
#include <asm/io.h>
#include <linux/cpumask.h>
#include <linux/smp.h>
#include <linux/log2.h>
#include <linux/irq.h>
#include <linux/pci.h>
#include <linux/msi.h>
#include <linux/platform_device.h>
#include <linux/irqnr.h>
```

```
#include <linux/types.h>
#include <linux/cdev.h>
#include <linux/irqdomain.h>
#include <linux/debugfs.h>
#include <linux/pci_regs.h>
#include <linux/irqdesc.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/of_irq.h>
#include <linux/delay.h>
#include <linux/irqchip/arm-gic-v3.h>
#include <vlx/htf-osal.h>
```

Data Structures

- struct _lpi_desc_t
- struct _htf_vgic_vdev_t
- struct _tc_desc_t

Functions

- static int _htf_vgic_vdev_setup (htf_gtest_id_t tid, void ∗cookie)

  Setup test execution environment and resources.
- static htf_status_t _htf_vgic_vdev_run (htf_gtest_id_t tid, void ∗cookie, char ∗args, size_t size)

  HTF vGIC virtual device test case `Test ID 115002`
- static void _htf_vgic_vdev_cleanup (htf_gtest_id_t tid, void ∗cookie)

  Cleanup test execution environment and resources.

9.6.1 Detailed Description

This file contains a HTF virtual device driver for testing vGIC emulation.

RQM 115002 : HTF virtual device for testing vGIC emulation.

Date

2002-2022

Copyright

9.6.2 Data Structure Documentation

9.6.2.1 struct _lpi_desc_t

Definition at line 87 of file htf-vgic-vdev.c.

9.6.2.2 struct _htf_vgic_vdev_t

Definition at line 95 of file htf-vgic-vdev.c.

9.6.2.3 struct _tc_desc_t

Definition at line 1869 of file htf-vgic-vdev.c.

9.6.3 Function Documentation

9.6.3.1 _htf_vgic_vdev_setup()

```
static int _htf_vgic_vdev_setup (
            htf_gtest_id_t tid,
            void * cookie )  [static]
```

Setup test execution environment and resources.

Parameters

tid	The test identifier as provided by HTF.
cookie	The test's private data passed back.

Return values

TRUE(1)	Setup was successful.
FALSE(0)	Setup failed.

Definition at line 1407 of file htf-vgic-vdev.c.

9.6.3.2 _htf_vgic_vdev_run()

```
static htf_status_t _htf_vgic_vdev_run (
            htf_gtest_id_t tid,
            void * cookie,
            char * args,
            size_t size )  [static]
```

HTF vGIC virtual device test case `Test ID 115002`

Parameters

tid	The test identifier provided by HTF.
cookie	The test private data passed back.
args	The provided options string.
size	The size of the provided options string.

Returns

The test execution status.

Precondition

This test is executed in the Hypervisor Test Framework (HTF).

Postcondition

There is no specific post condition for this test.

Test This test case provides

Definition at line 1980 of file htf-vgic-vdev.c.

9.6.3.3 _htf_vgic_vdev_cleanup()

```
static void _htf_vgic_vdev_cleanup (
            htf_gtest_id_t tid,
            void * cookie )  [static]
```

Cleanup test execution environment and resources.

This test does not require any cleanup.

Parameters

tid	The test identifier as provided by HTF.
cookie	The test private data.

Definition at line 2101 of file htf-vgic-vdev.c.

9.7 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/svec.h File Reference

Typedefs

- typedef void(∗ svec_hdl_t) (struct svec_event_handle ∗handle, void ∗cookie)

  Prototype of an event handler function.

Functions

- struct svec ∗ svec_lookup (const char ∗name)

    Finds a Simple Virtual Event Controller device.
- struct svec_event_handle ∗ svec_evt_attach (struct svec ∗svec, unsigned int event, svec_hdl_t handler, void ∗cookie, bool masked)

    Attaches (i.e. associates) a couple (handler, cookie) to an event.
- int svec_evt_detach (struct svec_event_handle ∗handle)

    Detaches (i.e. dissociates) an event-handle from an event.
- int svec_evt_mask (struct svec_event_handle ∗handle)

    Masks (i.e. disables) an event-handle.
- int svec_evt_unmask (struct svec_event_handle ∗handle)

    Unmasks (i.e. enables) an event-handle.
- unsigned int svec_evt_event (struct svec_event_handle ∗handle)

    get the event identifier to which the handle is attached.
- struct svec ∗ svec_evt_svec (struct svec_event_handle ∗handle)

    get the svec-device to which the handle is attached.

9.7.1 Typedef Documentation

9.7.1.1 svec_hdl_t

```
typedef void(* svec_hdl_t) (struct svec_event_handle *handle, void *cookie)
```

Prototype of an event handler function.

Parameters

handle	of the handler being executed.
cookie	pointer provided at handler attachment.

Definition at line 48 of file svec.h.

9.7.2 Function Documentation

9.7.2.1 svec_lookup()

```
struct svec* svec_lookup (
            const char * name )
```

Finds a Simple Virtual Event Controller device.

The svec-device is referenced to by its name as it appears in the device DT node "info" property. Multiple opens of the same device are allowed. Can be called from any context (task or interrupt).

Parameters

name	the name of the device.

Returns

An opaque pointer which uniquely identifies the svec-device instance. This value is to be provided as a parameter in any subsequent call to svec_evt_attach(), or NULL if none svec-device instance was found.

Definition at line 453 of file svec.c.

9.7.2.2 svec_evt_attach()

```
struct svec_event_handle* svec_evt_attach (
            struct svec * svec,
            unsigned int event,
            svec_hdl_t handler,
            void * cookie,
            bool masked )
```

Attaches (i.e. associates) a couple (handler, cookie) to an event.

Such an attachment is so-called an event-handle. Such an event-handle is provided with a masked status. If the event-handle is unmasked, the handler function is called with the cookie provided as a parameter upon every occurrence of the associated event. The masked parameter specifies if the event-handle is to be initially masked. The occurrence of an event with no attached event-handle is lost. The occurrence of an event for a masked event-handle is memorized, the event is so-called pending for this event-handle. One pending event at most can be memorized per event-handle. Unmasking an event-handle that has a pending event immediately triggers the call of the associated handler. An event-handle identifier is returned to the caller, which uniquely identifies the event-handle in any subsequent primitive call. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

svec	an opaque pointer value which identifies the svec-device.
event	identifier of the event.
handler	the handler function to be associated with the event.
cookie	an opaque pointer value to be provided unchanged as a parameter of the handler function call.
masked	indicates if the handle is masked initially.

Returns

An opaque pointer value in the ERR_PTR() format which uniquely identifies the event-handle. This value is to be provided as a parameter in any subsequent call to svec_evt_detach(), svec_evt_mask(), and svec_evt_unmask().

Definition at line 296 of file svec.c.

9.7.2.3 svec_evt_detach()

```
int svec_evt_detach (
            struct svec_event_handle * handle )
```

Detaches (i.e. dissociates) an event-handle from an event.

A detached event-handle does no longer has its associated handler called upon occurrence of the event. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

an errno value which indicates the operation success or failure:

- 0: success.
- a negative value: failure.

Definition at line 341 of file svec.c.

9.7.2.4 svec_evt_mask()

```
int svec_evt_mask (
            struct svec_event_handle * handle )
```

Masks (i.e. disables) an event-handle.

During the period of time an event-handle is masked, the associated handler is no longer called upon the event occurrence, the event is memorized as pending instead, until the event-handle is unmasked again. Masking/unmasking sequences can be nested. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

an errno value which indicates the operation success or failure:

- 0: success.
- a negative value: failure.

Definition at line 375 of file svec.c.

9.7.2.5 svec_evt_unmask()

```
int svec_evt_unmask (
            struct svec_event_handle * handle )
```

Unmasks (i.e. enables) an event-handle.

Enables the call of the handler associated with this event upon occurrence. If the event was pending for this event-handle at the time of unmasking, the associated handler is called. Masking/unmasking sequences can be nested. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

an errno value which indicates the operation success or failure:

- success: 0.
- failure: a negative errno value.

Definition at line 413 of file svec.c.

9.7.2.6 svec_evt_event()

```
unsigned int svec_evt_event (
            struct svec_event_handle * handle )
```

get the event identifier to which the handle is attached.

Can be called from any context (task or interrupt). In practice this function is mostly important to get the svec-device instance from within the handler context in the purpose of calling svec_evt_attach() from this handler.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

the event identifier

Definition at line 120 of file svec.c.

9.7.2.7 svec_evt_svec()

```
struct svec* svec_evt_svec (
          struct svec_event_handle * handle )
```

get the svec-device to which the handle is attached.

Can be called from any context (task or interrupt). In practice this function is mostly important to get the event from the within handler context in the purpose of calling svec_evt_attach() from this handler.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

the svec-device

Definition at line 126 of file svec.c.

9.8 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vdmaheap/uapi/vdmaheap↩_stats_uapi.h File Reference

virtual dmaheap (vdmaheap) driver - userspace stats API

```
#include <linux/ioctl.h>
#include <linux/types.h>
```

Data Structures

- struct vdmaheap_stats_dev
  Statistics command's parameters. More...
- struct vdmaheap_stats_vrpc
- union vdmaheap_stats_res
- struct vdmaheap_stats_arg
- union vdmaheap_stats_data

Macros

- #define VDMAHEAP_IOC_STATS _IOWR(VDMAHEAP_IOC_MAGIC, 55, union vdmaheap_stats_data)
  Driver statistics command.

Enumerations

- enum vdmaheap_stats_type {
  VDMAHEAP_STATS_DEV,
  VDMAHEAP_STATS_VRPC_BE,
  VDMAHEAP_STATS_VRPC_FE }

9.8.1 Detailed Description

virtual dmaheap (vdmaheap) driver - userspace stats API

This file defines the interface exposed by the vdmaheap driver to userspace clients for the purpose of retrieving statistics related to dmaheap buffers import/export.

The sections below defines the concepts and the objects handled by the statistics feature. Please refer to vdmaheap architectural design document for a more detailed description.

trusted vs. untrusted client.
An *untrusted client* is required to provide valid credentials upon each import request. Credentials provided at import time are checked against the ones produced at export time through a vRPC import request/response message exchange between the importer's and exporter's sides. If the credentials validity could not be established, the import request fails.
A *trusted client* is not required to provided valid credentials upon import requests. For such a client, a vRPC import request/response message exchange is also performed, but it only intends to check that the buffer is still exported (credential validity is not checked). If the buffer does not exist or no longer exists, because it was unexported for example, the import request fails.

strict import semantic.
When *strict import semantic* is enforced, a vRPC import request/response exchange is performed to check existence and/or credential validity on the exporter's side. It comes that if the buffer was unexported in the meantime, the import request fails. When the *strict import semantic* constraint is relaxed, and if the importer client is trusted, a re-import of a known physical dma_buf is processed locally on the importer's side, thus saving a vRPC exchange. The *strict import semantic* constraint relaxing is the sine qua non condition for vRPC exchanges optimization.

local vs. remote import.
A *local import* relates to a physical dma_buf object located on the importer's domain. A non-local import is a *remote import*.

cached vs. non-cached import.
A *cached import* originates from a client whose "requested grace" attribute is non-null, and is destined to a client whose "accepted grace" attribute is also non-null. When these conditions are met, the virtual dma_buf object and its physical counterpart have their lifespan extended by an amount of time defined by the negotiated grace delay (i.e. min(requested grace, accepted grace)). The grace delay starts running as soon as the virtual dma_buf object is no longer in use by any client.

physical vs. virtual dma_buf object.
A *physical dma_buf object* is the original exported dma_buf data structure. It only exists on an exporter's side and results from an allocation performed on /dev/dmaheap/xxx. It is unknown of vdmaheap until it is exported. It is associated with an export object in a 1-1 relationship. It exists as long as one of the conditions below is true:

- it still has an associated export object,
- it still has its file descriptor opened, it is destroyed as soon as these conditions are false.

  A *virtual dma_buf object* is a representation of a physical dma_buf object on an importer's side. A physical dma_buf object is usually represented by a single virtual dma_buf per domain, but there may be several virtual dma_buf objects representing the same physical object in the system (i.e. on several domains). It comes that physical and virtual dma_buf objects are associated in a 1-n relationship. It is created upon remote import of its physical counterpart object, and is associated with an import object. It exists as long as it is in use by a client (i.e. it still has one of the associated file descriptors opened), it is destroyed as soon after.

For a *non-cached import*, the import and virtual dma_buf objects are associated in a 1-1 relationship, and their lifespan perfectly match. It comes that dma_buf and import object counts also match perfectly in that case.

For a *cached import* things are a bit more subtle: Upon the first import, 2 virtual dma_buf objects are created. A principal one is provided to the requesting client, an additional copy is inserted in the cache. The principal object exists as long as it is in-use by the client, and is destroyed as soon after. Upon destruction of the principal object, the grace delay is started. The copy exists as long as one of the following condition is true:

- the principal virtual dma_buf object is in-use by a client (i.e. there remains a file descriptor opened),
- the grace delay is running,
  As soon after these conditions are false, the copy is destroyed. The copy in the cache serves as a template to re-create locally on the fly a principal virtual dma_buf object in the case of a re-import, without generating any vRPC traffic. It comes that in the case of cached imports, there may be more virtual dma_buf objects created than import ones.

note: In the context of vdmaheap statistics, a dma_buf object designates the virtual object.

New vs. known vs. same physical dma_buf objects.
A known dma_buf object is one that is present in the vdmaheap entity registry. As soon as a dma_buf object stops existing in vdmaheap, it is removed from the registry and becomes unknown, note that it may keep existing outside of vdmaheap. As long as a dma_buf is known, any dma_buf object with the same virtual address can unambiguously be considered as being the same object: as still being referenced in the registry, it could not, in any way have been freed and re-allocated. Conversely, trying to remember virtual addresses of dma_buf objects which have exited the registry with the intention to recognize them upon a future export is not a reliable method (objects might have been freed and re-allocated).

export object.
An *export object* only exists on the exporter's side, and is created the first time a physical dma_buf object is exported by a client. Multiple exports of the same physical dma_buf object result into the creation of a single export object. An export object exists as long as one of the conditions below is true:

- it still is unexported, or if exported multiple times, still has an unexported instance,
- it still has an associated virtual dma_buf object,

import object.
An *import object* only exists on the importer's side, and only in the case of a remote import. The local import of a dma_buf does not create a new import object. Multiple imports of the same physical dma_buf object result into the creation of a single import object. An import object is associated with a virtual dma_buf object in-use by a client, or a virtual dma_buf object copy residing in the cache. An import object exists as long as one of the conditions below is true:

- it's associated virtual dma_buf is still is in-use by a client (i.e. still has a file descriptor open),
- it has a virtual dma_buf object residing in the cache,

Idle state.
vdmaheap is in idle state if it fulfills the following conditions:

- it doesn't have any pending transaction (i.e. any export was closed by a a matching unexport),
- it does not have any virtual dma_buf object in-use by a client (i.e. all their file descriptor closed),
- all virtual dma_buf objects residing in the cache have been evicted (i.e. have their grace delay expired),

9.9 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vdmaheap/uapi/vdmaheap_test_uapi.h File Reference

virtual DMAHEAP (vDMAHEAP) driver - userspace test API

```
#include <linux/ioctl.h>
#include <linux/types.h>
#include "vdmaheap_uapi.h"
```

Data Structures

- struct vdmaheap_ioc_arg_set_trusted_t
- struct vdmaheap_ioc_arg_set_strict_import_t
- struct vdmaheap_ioc_arg_set_grace_t
- union vdmaheap_ioc_test_arg_t
- union vdmaheap_ioc_test_res_t
- struct vdmaheap_test_data
- struct vdmaheap_import_mult_data_in
- struct vdmaheap_import_mult_data_out
- struct vdmaheap_import_mult_data
- union vdmaheap_test_data.__unnamed__
- union vdmaheap_import_mult_data.__unnamed__

Macros

- #define VDMAHEAP_IOC_TEST _IOWR(VDMAHEAP_IOC_MAGIC, 56, struct vdmaheap_test_data)

  Driver test command.
- #define VDMAHEAP_IOC_IMPORT_MULTIPLE _IOWR(VDMAHEAP_IOC_MAGIC, 57, struct vdmaheap_import_mult_data)

  Multiple buffers import command.
- #define VDMAHEAP_IOC_IMPORT_MULTIPLE_MAX 10

  Import command's argument.

Enumerations

- enum vdmaheap_ioc_test_command_t

  test command's parameters

9.9.1 Detailed Description

virtual DMAHEAP (vDMAHEAP) driver - userspace test API

This file defines the interface exposed by the vDMAHEAP driver to userspace clients for the purpose of testing.

9.9.2 Data Structure Documentation

9.9.2.1 struct vdmaheap_ioc_arg_set_trusted_t

Definition at line 58 of file vdmaheap_test_uapi.h.

9.9.2.2 struct vdmaheap_ioc_arg_set_strict_import_t

Definition at line 61 of file vdmaheap_test_uapi.h.

9.9.2.3 struct vdmaheap_ioc_arg_set_grace_t

Definition at line 64 of file vdmaheap_test_uapi.h.

9.9.2.4 union vdmaheap_ioc_test_arg_t

Definition at line 69 of file vdmaheap_test_uapi.h.

9.9.2.5 union vdmaheap_ioc_test_res_t

Definition at line 77 of file vdmaheap_test_uapi.h.

9.9.2.6 struct vdmaheap_test_data

Definition at line 83 of file vdmaheap_test_uapi.h.

Data Fields

vdmaheap_ioc_test_command_t	command	[in] test command
union vdmaheap_test_data	__unnamed__	

9.9.2.7 struct vdmaheap_import_mult_data_in

Definition at line 108 of file vdmaheap_test_uapi.h.

Data Fields

vdmaheap_buffer_gid_t	gids[VDMAHEAP_IOC_IMPORT_MULTIPLE_MAX]	[in] global buffer identifier
uuid_t	creds[VDMAHEAP_IOC_IMPORT_MULTIPLE_MAX]	[in] buffer credentials

9.9.2.8 struct vdmaheap_import_mult_data_out

Definition at line 113 of file vdmaheap_test_uapi.h.

Data Fields

__s32	fd	[out] imported DMAHEAP buffer's file descriptor
__u32	size	[out] buffer size

9.9.2.9 struct vdmaheap_import_mult_data

Definition at line 117 of file vdmaheap_test_uapi.h.

9.9.2.10 union vdmaheap_test_data.__unnamed__

Definition at line 85 of file vdmaheap_test_uapi.h.

Data Fields

vdmaheap_ioc_test_arg_t	arg	[in] command args
vdmaheap_ioc_test_res_t	res	[out] command res

9.9.2.11 union vdmaheap_import_mult_data.__unnamed__

Definition at line 121 of file vdmaheap_test_uapi.h.

9.9.3 Macro Definition Documentation

9.9.3.1 VDMAHEAP_IOC_TEST

```
#define VDMAHEAP_IOC_TEST _IOWR(VDMAHEAP_IOC_MAGIC, 56, struct vdmaheap_test_data)
```

Driver test command.

This command allows to control vDMAHEAP behavior under test conditions.

Definition at line 42 of file vdmaheap_test_uapi.h.

9.9.3.2 VDMAHEAP_IOC_IMPORT_MULTIPLE

```
#define VDMAHEAP_IOC_IMPORT_MULTIPLE _IOWR(VDMAHEAP_IOC_MAGIC, 57, struct vdmaheap_import_mult_data)
```

Multiple buffers import command.

This command enables the vDMAHEAP client to import a DMAHEAP buffer. The import operation provides the client with a file descriptor pointing at the imported DMAHEAP buffer. This file descriptor can be used to map the buffer in userspace and to perform memory synchronization operations on the buffer.

Definition at line 99 of file vdmaheap_test_uapi.h.

9.9.3.3 VDMAHEAP_IOC_IMPORT_MULTIPLE_MAX

```
#define VDMAHEAP_IOC_IMPORT_MULTIPLE_MAX 10
```

Import command's argument.

This data structure is used as the argument of the VDMAHEAP_IOC_IMPORT command.

Definition at line 107 of file vdmaheap_test_uapi.h.

9.9.4 Enumeration Type Documentation

9.9.4.1 vdmaheap_ioc_test_command_t

```
enum vdmaheap_ioc_test_command_t
```

test command's parameters

This data structure is used as the argument of the VDMAHEAP_IOC_TEST command.

Definition at line 49 of file vdmaheap_test_uapi.h.

9.10 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vdmaheap/uapi/vdmaheap_uapi.h File Reference

virtual DMAHEAP (vDMAHEAP) driver - userspace API

```
#include <linux/ioctl.h>
#include <linux/types.h>
#include <linux/uuid.h>
#include <uapi/linux/dma-heap.h>
#include "vdmaheap_stats_uapi.h"
```

Data Structures

- struct uuid_t
  Buffer credentials. More...
- struct vdmaheap_version_data
  API version handshake command's argument. More...
- struct vdmaheap_export_data
  Export command's argument. More...
- struct vdmaheap_unexport_data
  Unexport command's argument. More...
- struct vdmaheap_import_data
  Import command's argument. More...
- struct vdmaheap_info_data
  Info command's parameters. More...
- struct vdmaheap_link_state
  Info command's parameters. More...

Macros

- #define VDMAHEAP_IOC_MAGIC DMA_HEAP_IOC_MAGIC

  driver commands's magic number
- #define UUID_SIZE 16

  Length of a UUID's byte array.
- #define VDMAHEAP_VERSION_0 0
- #define VDMAHEAP_VERSION_1 1
- #define VDMAHEAP_VERSION_2 2
- #define VDMAHEAP_VERSION_3 3
- #define VDMAHEAP_VERSION_4 4
- #define VDMAHEAP_VERSION VDMAHEAP_VERSION_4

  Userspace API's current version number.
- #define VDMAHEAP_CAP_SECURE (1U << 0)
- #define VDMAHEAP_CAP_LOCAL (1U << 1)
- #define VDMAHEAP_CAP_REMOTE (1U << 2)
- #define VDMAHEAP_CAP_VBB (1U << 3)
- #define VDMAHEAP_CAP_STRICT_IMPORT
- #define VDMAHEAP_IOC_VERSION _IOWR(VDMAHEAP_IOC_MAGIC, 54, struct vdmaheap_version_data)

  API version handshake command.
- #define VDMAHEAP_IOC_EXPORT _IOWR(VDMAHEAP_IOC_MAGIC, 50, struct vdmaheap_export_data)

  Buffer export command.
- #define VDMAHEAP_IOC_UNEXPORT _IOWR(VDMAHEAP_IOC_MAGIC, 51, struct vdmaheap_unexport_data)

  Buffer unexport command.
- #define VDMAHEAP_IOC_IMPORT _IOWR(VDMAHEAP_IOC_MAGIC, 52, struct vdmaheap_import_data)

  Buffer import command.
- #define VDMAHEAP_IOC_INFO _IOWR(VDMAHEAP_IOC_MAGIC, 53, struct vdmaheap_info_data)

  Buffer information command.
- #define VDMAHEAP_IOC_LINK_STATE _IOWR(VDMAHEAP_IOC_MAGIC, 55, struct vdmaheap_info_data)

  Read/wait for vDMAHEAP connection with its peer backends.

Typedefs

- typedef __u32 vdmaheap_buffer_gid_t

  Global buffer identifier.

Functions

- static int vdmaheap_gid_origin (vdmaheap_buffer_gid_t gid)

  Buffer origin domain.

9.10.1 Detailed Description

virtual DMAHEAP (vDMAHEAP) driver - userspace API

This file defines the interface exposed by the vDMAHEAP driver to userspace clients for the purpose of sharing DMAHEAP buffers between processes.

9.11 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vdmaheap/uapi/vdmaheap↩ _vbb_uapi.h File Reference

Data Structures

- struct vbb_buffer

  structure to exchange data between user and kernel spaces. More...

Macros

- #define VBB_IOC_CREATE_BUFID _IOWR(VBB_IOC_MAGIC, 1, struct vbb_buffer)

  exports a dmabuf.
- #define VBB_IOC_DESTROY_BUFID _IOWR(VBB_IOC_MAGIC, 2, struct vbb_buffer)

  unexports a dmabuf.
- #define VBB_IOC_BUFID_TO_FD _IOWR(VBB_IOC_MAGIC, 3, struct vbb_buffer)

  imports a dmabuf.

9.11.1 Detailed Description

include/linux/vbb.h

9.11.2 Data Structure Documentation

9.11.2.1 struct vbb_buffer

structure to exchange data between user and kernel spaces.

this data structure is used in the vbb-user-API.

Definition at line 17 of file vdmaheap_vbb_uapi.h.

Data Fields

int	fd	samsung_dma_buffer fd ion_buffer fd returned by ion_alloc()
size_t	length	buffer length
unsigned int	buf_id	unique buffer GID returned by export()
unsigned char	no_cached	flag for cache/non-cache mapping for domU

9.11.3 Macro Definition Documentation

9.11.3.1 VBB_IOC_CREATE_BUFID

```
#define VBB_IOC_CREATE_BUFID _IOWR(VBB_IOC_MAGIC, 1, struct vbb_buffer)
```

exports a dmabuf.

takes the dmabuf fd as input parameter, returns the GID assigned to the dmabuf.

Definition at line 36 of file vdmaheap_vbb_uapi.h.

9.11.3.2 VBB_IOC_DESTROY_BUFID

```
#define VBB_IOC_DESTROY_BUFID _IOWR(VBB_IOC_MAGIC, 2, struct vbb_buffer)
```

unexports a dmabuf.

takes the GID as input parameter. Unlike regular user-API, this function does not take credential as input parameter, this may be a security issue.

Definition at line 45 of file vdmaheap_vbb_uapi.h.

9.11.3.3 VBB_IOC_BUFID_TO_FD

```
#define VBB_IOC_BUFID_TO_FD _IOWR(VBB_IOC_MAGIC, 3, struct vbb_buffer)
```

imports a dmabuf.

takes the GID as input parameter. Unlike regular user-API, this function does not take credential as input parameter, this may be a security issue.

Definition at line 54 of file vdmaheap_vbb_uapi.h.

9.12 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vfence2/uapi/vfence2↩_uapi.h File Reference

virtual DMA fence v2 (vFence2) driver - user space API

```
#include <linux/ioctl.h>
#include <linux/types.h>
```

Data Structures

- struct vfence2_export_data

  Export command's argument. More...
- struct vfence2_unexport_data

  Unexport command's argument. More...
- struct vfence2_import_data

  Import command's argument. More...

Macros

- #define VFENCE2_ID_NONE 0

  Non-existing DMA fence identifier.
- #define VFENCE2_IOC_MAGIC 'F'

  driver command magic number
- #define VFENCE2_IOC_EXPORT _IOWR(VFENCE2_IOC_MAGIC, 50, struct vfence2_export_data)

  DMA fence export command.
- #define VFENCE2_IOC_UNEXPORT _IOWR(VFENCE2_IOC_MAGIC, 51, struct vfence2_unexport_data)

  DMA fence unexport command.
- #define VFENCE2_IOC_IMPORT _IOWR(VFENCE2_IOC_MAGIC, 52, struct vfence2_import_data)

  DMA fence import command.

Typedefs

- typedef __u32 vfence2_id_t

  Exported DMA fence identifier.

9.12.1 Detailed Description

virtual DMA fence v2 (vFence2) driver - user space API

This file defines the interface exposed by the vFence2 driver to user space clients for the purpose of sharing DMA fences between processes.

9.13 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vion/uapi/vion_stats_uapi.h File Reference

virtual ION (vION) driver - userspace stats API

```
#include <linux/ioctl.h>
#include <linux/types.h>
```

Data Structures

- struct vion_stats_dev

  Statistics command's parameters. More...
- struct vion_stats_vrpc
- union vion_stats_res
- struct vion_stats_arg
- union vion_stats_data

Macros

- #define VION_IOC_STATS _IOWR(VION_IOC_MAGIC, 55, union vion_stats_data)

  Driver statistics command.

Enumerations

- enum vion_stats_type {
  VION_STATS_DEV,
  VION_STATS_VRPC_BE,
  VION_STATS_VRPC_FE }

9.13.1 Detailed Description

virtual ION (vION) driver - userspace stats API

This file defines the interface exposed by the vION driver to userspace clients for the purpose of retrieving statistics related to ION buffers import/export.

The sections below defines the concepts and the objects handled by the statistics feature. Please refer to vION architectural design document for a more detailed description.

trusted vs. untrusted client.
An *untrusted client* is required to provide valid credentials upon each import request. Credentials provided at import time are checked against the ones produced at export time through a vRPC import request/response message exchange between the importer's and exporter's sides. If the credentials validity could not be established, the import request fails.
A *trusted client* is not required to provided valid credentials upon import requests. For such a client, a vRPC import request/response message exchange is also performed, but it only intends to check that the buffer is still exported (credential validity is not checked). If the buffer does not exist or no longer exists, because it was unexported for example, the import request fails.

strict import semantic.
When *strict import semantic* is enforced, a vRPC import request/response exchange is performed to check existence and/or credential validity on the exporter's side. It comes that if the buffer was unexported in the meantime, the import request fails. When the *strict import semantic* constraint is relaxed, and if the importer client is trusted, a re-import of a known physical dma_buf is processed locally on the importer's side, thus saving a vRPC exchange. The *strict import semantic* constraint relaxing is the sine qua non condition for vRPC exchanges optimization.

local vs. remote import.
A *local import* relates to a physical dma_buf object located on the importer's domain. A non-local import is a *remote import*.

cached vs. non-cached import.
A *cached import* originates from a client whose "requested grace" attribute is non-null, and is destined to a client whose "accepted grace" attribute is also non-null. When these conditions are met, the virtual dma_buf object and its physical counterpart have their lifespan extended by an amount of time defined by the negotiated grace delay (i.e. min(requested grace, accepted grace)). The grace delay starts running as soon as the virtual dma_buf object is no longer in use by any client.

physical vs. virtual dma_buf object.
A *physical dma_buf object* is the original exported dma_buf data structure. It only exists on an exporter's side and results from an allocation performed on /dev/ion. It is unknown of vION until it is exported. It is associated with an export object in a 1-1 relationship. It exists as long as one of the conditions below is true:

- it still has an associated export object,

- it still has its file descriptor opened, it is destroyed as soon as these conditions are false.

  A *virtual dma_buf object* is a representation of a physical dma_buf object on an importer's side. A physical dma_buf object is usually represented by a single virtual dma_buf per domain, but there may be several virtual dma_buf objects representing the same physical object in the system (i.e. on several domains). It comes that physical and virtual dma_buf objects are associated in a 1-n relationship. It is created upon remote import of its physical counterpart object, and is associated with an import object. It exists as long as it is in use by a client (i.e. it still has one of the associated file descriptors opened), it is destroyed as soon after.

For a *non-cached import*, the import and virtual dma_buf objects are associated in a 1-1 relationship, and their lifespan perfectly match. It comes that dma_buf and import object counts also match perfectly in that case.

For a *cached import* things are a bit more subtle: Upon the first import, 2 virtual dma_buf objects are created. A principal one is provided to the requesting client, an additional copy is inserted in the cache. The principal object exists as long as it is in-use by the client, and is destroyed as soon after. Upon destruction of the principal object, the grace delay is started. The copy exists as long as one of the following condition is true:

- the principal virtual dma_buf object is in-use by a client (i.e. there remains a file descriptor opened),
- the grace delay is running,
  As soon after these conditions are false, the copy is destroyed. The copy in the cache serves as a template to re-create locally on the fly a principal virtual dma_buf object in the case of a re-import, without generating any vRPC traffic. It comes that in the case of cached imports, there may be more virtual dma_buf objects created than import ones.

note: In the context of vION statistics, a dma_buf object designates the virtual object.

New vs. known vs. same physical dma_buf objects.
A known dma_buf object is one that is present in the vION entity registry. As soon as a dma_buf object stops existing in vION, it is removed from the registry and becomes unknown, note that it may keep existing outside of vION. As long as a dma_buf is known, any dma_buf object with the same virtual address can unambiguously be considered as being the same object: as still being referenced in the registry, it could not, in any way have been freed and re-allocated. Conversely, trying to remember virtual addresses of dma_buf objects which have exited the registry with the intention to recognize them upon a future export is not a reliable method (objects might have been freed and re-allocated).

export object.
An *export object* only exists on the exporter's side, and is created the first time a physical dma_buf object is exported by a client. Multiple exports of the same physical dma_buf object result into the creation of a single export object. An export object exists as long as one of the conditions below is true:

- it still is unexported, or if exported multiple times, still has an unexported instance,
- it still has an associated virtual dma_buf object,

import object.
An *import object* only exists on the importer's side, and only in the case of a remote import. The local import of a dma_buf does not create a new import object. Multiple imports of the same physical dma_buf object result into the creation of a single import object. An import object is associated with a virtual dma_buf object in-use by a client, or a virtual dma_buf object copy residing in the cache. An import object exists as long as one of the conditions below is true:

- it's associated virtual dma_buf is still is in-use by a client (i.e. still has a file descriptor open),
- it has a virtual dma_buf object residing in the cache,

Idle state.
vION is in idle state if it fulfills the following conditions:

- it doesn't have any pending transaction (i.e. any export was closed by a matching unexport),
- it does not have any virtual dma_buf object in-use by a client (i.e. all their file descriptor closed),
- all virtual dma_buf objects residing in the cache have been evicted (i.e. have their grace delay expired),

9.14 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vion/uapi/vion_test_uapi.h File Reference

virtual ION (vION) driver - userspace test API

```
#include <linux/ioctl.h>
#include <linux/types.h>
#include "vion_uapi.h"
```

Data Structures

- struct vion_ioc_arg_set_trusted_t
- struct vion_ioc_arg_set_strict_import_t
- struct vion_ioc_arg_set_grace_t
- union vion_ioc_test_arg_t
- union vion_ioc_test_res_t
- struct vion_test_data
- struct vion_import_mult_data_in
- struct vion_import_mult_data_out
- struct vion_import_mult_data
- union vion_test_data.__unnamed__
- union vion_import_mult_data.__unnamed__

Macros

- #define VION_IOC_TEST _IOWR(VION_IOC_MAGIC, 56, struct vion_test_data)

    Driver test command.
- #define VION_IOC_IMPORT_MULTIPLE _IOWR(VION_IOC_MAGIC, 57, struct vion_import_mult_data)

    Multiple buffers import command.
- #define VION_IOC_IMPORT_MULTIPLE_MAX 10

    Import command's argument.

Enumerations

- enum vion_ioc_test_command_t

    test command's parameters

9.14.1 Detailed Description

virtual ION (vION) driver - userspace test API

This file defines the interface exposed by the vION driver to userspace clients for the purpose of testing.

9.14.2 Data Structure Documentation

9.14.2.1 struct vion_ioc_arg_set_trusted_t

Definition at line 58 of file vion_test_uapi.h.

9.14.2.2 struct vion_ioc_arg_set_strict_import_t

Definition at line 61 of file vion_test_uapi.h.

9.14.2.3 struct vion_ioc_arg_set_grace_t

Definition at line 64 of file vion_test_uapi.h.

9.14.2.4 union vion_ioc_test_arg_t

Definition at line 69 of file vion_test_uapi.h.

9.14.2.5 union vion_ioc_test_res_t

Definition at line 77 of file vion_test_uapi.h.

9.14.2.6 struct vion_test_data

Definition at line 83 of file vion_test_uapi.h.

Data Fields

vion_ioc_test_command_t	command	[in] test command
union vion_test_data	__unnamed__	

9.14.2.7 struct vion_import_mult_data_in

Definition at line 108 of file vion_test_uapi.h.

Data Fields

vion_buffer_gid_t	gids[VION_IOC_IMPORT_MULTIPLE_MAX]	[in] global buffer identifier
uuid_t	creds[VION_IOC_IMPORT_MULTIPLE_MAX]	[in] buffer credentials

9.14.2.8 struct vion_import_mult_data_out

Definition at line 113 of file vion_test_uapi.h.

Data Fields

__s32	fd	[out] imported ION buffer's file descriptor
__u32	size	[out] buffer size

9.14.2.9 struct vion_import_mult_data

Definition at line 117 of file vion_test_uapi.h.

9.14.2.10 union vion_test_data.__unnamed__

Definition at line 85 of file vion_test_uapi.h.

Data Fields

vion_ioc_test_arg_t	arg	[in] command args
vion_ioc_test_res_t	res	[out] command res

9.14.2.11 union vion_import_mult_data.__unnamed__

Definition at line 121 of file vion_test_uapi.h.

9.14.3 Macro Definition Documentation

9.14.3.1 VION_IOC_TEST

```
#define VION_IOC_TEST _IOWR(VION_IOC_MAGIC, 56, struct vion_test_data)
```

Driver test command.

This command allows to control vION behavior under test conditions.

Definition at line 42 of file vion_test_uapi.h.

9.14.3.2 VION_IOC_IMPORT_MULTIPLE

```
#define VION_IOC_IMPORT_MULTIPLE _IOWR(VION_IOC_MAGIC, 57, struct vion_import_mult_data)
```

Multiple buffers import command.

This command enables the vION client to import a ION buffer. The import operation provides the client with a file descriptor pointing at the imported ION buffer. This file descriptor can be used to map the buffer in userspace and to perform memory synchronization operations on the buffer.

Definition at line 99 of file vion_test_uapi.h.

9.14.3.3 VION_IOC_IMPORT_MULTIPLE_MAX

```
#define VION_IOC_IMPORT_MULTIPLE_MAX 10
```

Import command's argument.

This data structure is used as the argument of the VION_IOC_IMPORT command.

Definition at line 107 of file vion_test_uapi.h.

9.14.4 Enumeration Type Documentation

9.14.4.1 vion_ioc_test_command_t

```
enum vion_ioc_test_command_t
```

test command's parameters

This data structure is used as the argument of the VION_IOC_TEST command.

Definition at line 49 of file vion_test_uapi.h.

9.15 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vion/uapi/vion_uapi.h File Reference

virtual ION (vION) driver - userspace API

```
#include <linux/ioctl.h>
#include <linux/types.h>
#include <linux/uuid.h>
#include "vion_stats_uapi.h"
```

Data Structures

- struct uuid_t

  Buffer credentials. More...
- struct vion_version_data

  API version handshake command's argument. More...
- struct vion_export_data

  Export command's argument. More...
- struct vion_unexport_data

  Unexport command's argument. More...
- struct vion_import_data

  Import command's argument. More...
- struct vion_info_data

  Info command's parameters. More...
- struct vion_link_state

  Info command's parameters. More...

Macros

- #define VION_IOC_MAGIC ION_IOC_MAGIC

  driver commands's magic number
- #define UUID_SIZE 16

  Length of a UUID's byte array.
- #define VION_VERSION_0 0
- #define VION_VERSION_1 1
- #define VION_VERSION_2 2
- #define VION_VERSION_3 3
- #define VION_VERSION_4 4
- #define VION_VERSION VION_VERSION_4

  Userspace API's current version number.
- #define VION_CAP_SECURE (1U << 0)
- #define VION_CAP_LOCAL (1U << 1)
- #define VION_CAP_REMOTE (1U << 2)
- #define VION_CAP_VBB (1U << 3)
- #define VION_CAP_STRICT_IMPORT
- #define VION_IOC_VERSION _IOWR(VION_IOC_MAGIC, 54, struct vion_version_data)

  API version handshake command.
- #define VION_IOC_EXPORT _IOWR(VION_IOC_MAGIC, 50, struct vion_export_data)

  Buffer export command.
- #define VION_IOC_UNEXPORT _IOWR(VION_IOC_MAGIC, 51, struct vion_unexport_data)

  Buffer unexport command.
- #define VION_IOC_IMPORT _IOWR(VION_IOC_MAGIC, 52, struct vion_import_data)

  Buffer import command.
- #define VION_IOC_INFO _IOWR(VION_IOC_MAGIC, 53, struct vion_info_data)

  Buffer information command.
- #define VION_IOC_LINK_STATE _IOWR(VION_IOC_MAGIC, 55, struct vion_info_data)

  Read/wait for vION connection with its peer backends.

Typedefs

- typedef __u32 vion_buffer_gid_t

  Global buffer identifier.

Functions

- static int vion_gid_origin (vion_buffer_gid_t gid)

  Buffer origin domain.

9.15.1 Detailed Description

virtual ION (vION) driver - userspace API

This file defines the interface exposed by the vION driver to userspace clients for the purpose of sharing ION buffers between processes.

9.16 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vion/uapi/vion_vbb_uapi.h File Reference

Data Structures

- struct vbb_buffer

  structure to exchange data between user and kernel spaces. More...

Macros

- #define VBB_IOC_CREATE_BUFID _IOWR(VBB_IOC_MAGIC, 1, struct vbb_buffer)

  exports a dmabuf.
- #define VBB_IOC_DESTROY_BUFID _IOWR(VBB_IOC_MAGIC, 2, struct vbb_buffer)

  unexports a dmabuf.
- #define VBB_IOC_BUFID_TO_FD _IOWR(VBB_IOC_MAGIC, 3, struct vbb_buffer)

  imports a dmabuf.

9.16.1 Detailed Description

include/linux/vbb.h

9.16.2 Data Structure Documentation

9.16.2.1 struct vbb_buffer

structure to exchange data between user and kernel spaces.

this data structure is used in the vbb-user-API.

Definition at line 17 of file vdmaheap_vbb_uapi.h.

Data Fields

int	fd	samsung_dma_buffer fd ion_buffer fd returned by ion_alloc()
size_t	length	buffer length
unsigned int	buf_id	unique buffer GID returned by export()
unsigned char	no_cached	flag for cache/non-cache mapping for domU

9.16.3 Macro Definition Documentation

9.16.3.1 VBB_IOC_CREATE_BUFID

```
#define VBB_IOC_CREATE_BUFID _IOWR(VBB_IOC_MAGIC, 1, struct vbb_buffer)
```

exports a dmabuf.

takes the dmabuf fd as input parameter, returns the GID assigned to the dmabuf.

Definition at line 36 of file vion_vbb_uapi.h.

9.16.3.2 VBB_IOC_DESTROY_BUFID

```
#define VBB_IOC_DESTROY_BUFID _IOWR(VBB_IOC_MAGIC, 2, struct vbb_buffer)
```

unexports a dmabuf.

takes the GID as input parameter. Unlike regular user-API, this function does not take credential as input parameter, this may be a security issue.

Definition at line 45 of file vion_vbb_uapi.h.

9.16.3.3 VBB_IOC_BUFID_TO_FD

```
#define VBB_IOC_BUFID_TO_FD _IOWR(VBB_IOC_MAGIC, 3, struct vbb_buffer)
```

imports a dmabuf.

takes the GID as input parameter. Unlike regular user-API, this function does not take credential as input parameter, this may be a security issue.

Definition at line 54 of file vion_vbb_uapi.h.

9.17 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-virtio.h File Reference

VLX VirtIO Control Plane API.

```
#include <linux/types.h>
```

Data Structures

- struct vlx_virtio_dev

    Representation of a VirtIO device. More...
- struct vlx_virtio_dev_cb

    VirtIO device-specific operations.

Enumerations

- enum vlx_virtio_api {
  VLX_VIRTIO_KERNEL,
  VLX_VIRTIO_USER,
  VLX_VIRTIO_VHOST }

Functions

- struct vlx_virtio_dev ∗ vlx_virtio_lookup_device (int vmid, const char ∗dev_name, enum vlx_virtio_api api)

  Lookup and claim a VirtIO device using the front-end VM id and the device name.
- void vlx_virtio_release_device (struct vlx_virtio_dev ∗dev)

  Releases a VirtIO device that was previously claimed through vlx_virtio_lookup_device.
- int vlx_virtio_register_device (struct vlx_virtio_dev ∗dev)

  Register a new device with the framework after initialization.
- void vlx_virtio_unregister_device (struct vlx_virtio_dev ∗dev)

  Tell the VLX VirtIO framework that the back-end driver is no longer willing to manage this device.
- int vlx_virtio_device_init_vqs (struct vlx_virtio_dev ∗dev, unsigned int nbr)

  Allocate and initialize the virtqueue descriptors (see vlx_virtio_dev::vqs) for a device.
- void vlx_virtio_device_destroy_vqs (struct vlx_virtio_dev ∗dev)

  Releases the device virtqueue descriptors (see vlx_virtio_dev::vqs).
- void vlx_virtio_config_changed (struct vlx_virtio_dev ∗dev)

  Notify the driver after changes in the device configuration space.
- u32 vlx_virtio_status_get (struct vlx_virtio_dev ∗dev)

  Get the current device status bits.
- void vlx_virtio_status_set (struct vlx_virtio_dev ∗dev, u32 status)

  Set the current device status bits.

9.17.1 Detailed Description

VLX VirtIO Control Plane API.

9.18 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/include/vlx/vlx-virtq.h File Reference

VLX VirtIO Data Plane API.

```
#include <linux/types.h>
#include <linux/uio.h>
#include <linux/virtio_ring.h>
```

Data Structures

- struct vlx_virtq

  Representation of a virtqueue.

Enumerations

- enum kick_policy {
  DEV_WORKER,
  VQ_WORKER,
  DIRECT_CALL }

Functions

- int vlx_vq_init_access (struct vlx_virtq ∗vq)
  Map the virtqueue descriptors and rings.
- void vlx_vq_stop_access (struct vlx_virtq ∗vq)
  Unmap the virtqueue descriptors and rings.
- bool vlx_vq_ring_ready (struct vlx_virtq ∗vq)
  Check whether the virtqueue is ready for I/O.
- bool vlx_vq_has_descs (struct vlx_virtq ∗vq)
  Check whether a virtqueue has available buffers.
- int vlx_vq_getchain (struct vlx_virtq ∗vq, uint16_t ∗pidx, struct kvec ∗iov, int n_iov, uint16_t ∗flags)
  Get the next available chain of descriptors on a virtqueue and put it into an I/O vector.
- void vlx_vq_retchain (struct vlx_virtq ∗vq)
  Return the last chain returned by vlx_vq_getchain back to the available ring.
- void vlx_vq_relchain (struct vlx_virtq ∗vq, uint16_t idx, uint32_t iolen)
  Return specified chain to the used ring, setting its I/O length to the provided value.
- void vlx_vq_endchains (struct vlx_virtq ∗vq, int used_all_avail)
  Driver has finished processing available chains and calling vlx_vq_relchain on each one.
- void vlx_vq_clear_used_ring_flags (struct vlx_virtq ∗vq)
  Helper function for clearing used ring flags.
- void vlx_vq_set_used_ring_flags (struct vlx_virtq ∗vq)
  Helper function for setting used ring flags.
- void vlx_vq_disable_notification (struct vlx_virtq ∗vq)
  Disable driver notifications when new available buffers are added to a virtqueue.
- bool vlx_vq_enable_notification (struct vlx_virtq ∗vq)
  Enable driver notifications when new buffers are available.

9.18.1 Detailed Description

VLX VirtIO Data Plane API.

9.19 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/svec.c File Reference

```
#include <linux/module.h>
#include <linux/bug.h>
#include <linux/printk.h>
#include <linux/workqueue.h>
#include <linux/err.h>
#include <linux/slab.h>
#include <linux/list.h>
#include <linux/string.h>
#include <linux/spinlock.h>
```

```
#include <linux/atomic.h>
#include <linux/kref.h>
#include <linux/preempt.h>
#include <linux/version.h>
#include <asm/bitops.h>
#include <asm/cmpxchg.h>
#include <nk/nkern.h>
#include <vlx/svec.h>
#include <common/nk/nksvec.h>
```

Data Structures

- struct svec_handle_list
- struct svec_handle_repo
- struct svec_pmem
- struct svec_pxirq
- struct svec
- struct svec_event_handle

Functions

- unsigned int svec_evt_event (struct svec_event_handle ∗handle)

  get the event identifier to which the handle is attached.
- struct svec ∗ svec_evt_svec (struct svec_event_handle ∗handle)

  get the svec-device to which the handle is attached.
- struct svec_event_handle ∗ svec_evt_attach (struct svec ∗svec, unsigned int event, svec_hdl_t handler, void ∗cookie, bool masked)

  Attaches (i.e. associates) a couple (handler, cookie) to an event.
- int svec_evt_detach (struct svec_event_handle ∗handle)

  Detaches (i.e. dissociates) an event-handle from an event.
- int svec_evt_mask (struct svec_event_handle ∗handle)

  Masks (i.e. disables) an event-handle.
- int svec_evt_unmask (struct svec_event_handle ∗handle)

  Unmasks (i.e. enables) an event-handle.
- struct svec ∗ svec_lookup (const char ∗name)

  Finds a Simple Virtual Event Controller device.

9.19.1 Data Structure Documentation

9.19.1.1 struct svec_handle_list

Definition at line 62 of file svec.c.

9.19.1.2 struct svec_handle_repo

Definition at line 67 of file svec.c.

9.19.1.3 struct svec_pmem

Definition at line 71 of file svec.c.

9.19.1.4 struct svec_pxirq

Definition at line 77 of file svec.c.

9.19.1.5 struct svec

Definition at line 82 of file svec.c.

9.19.1.6 struct svec_event_handle

Definition at line 102 of file svec.c.

9.19.2 Function Documentation

9.19.2.1 svec_evt_event()

```
unsigned int svec_evt_event (
            struct svec_event_handle * handle )
```

get the event identifier to which the handle is attached.

Can be called from any context (task or interrupt). In practice this function is mostly important to get the svec-device instance from within the handler context in the purpose of calling svec_evt_attach() from this handler.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

the event identifier

Definition at line 120 of file svec.c.

9.19.2.2 svec_evt_svec()

```
struct svec* svec_evt_svec (
            struct svec_event_handle * handle )
```

get the svec-device to which the handle is attached.

Can be called from any context (task or interrupt). In practice this function is mostly important to get the event from the within handler context in the purpose of calling svec_evt_attach() from this handler.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

the svec-device

Definition at line 126 of file svec.c.

9.19.2.3 svec_evt_attach()

```
struct svec_event_handle* svec_evt_attach (
            struct svec * svec,
            unsigned int event,
            svec_hdl_t handler,
            void * cookie,
            bool masked )
```

Attaches (i.e. associates) a couple (handler, cookie) to an event.

Such an attachment is so-called an event-handle. Such an event-handle is provided with a masked status. If the event-handle is unmasked, the handler function is called with the cookie provided as a parameter upon every occurrence of the associated event. The masked parameter specifies if the event-handle is to be initially masked. The occurrence of an event with no attached event-handle is lost. The occurrence of an event for a masked event-handle is memorized, the event is so-called pending for this event-handle. One pending event at most can be memorized per event-handle. Unmasking an event-handle that has a pending event immediately triggers the call of the associated handler. An event-handle identifier is returned to the caller, which uniquely identifies the event-handle in any subsequent primitive call. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

svec	an opaque pointer value which identifies the svec-device.
event	identifier of the event.
handler	the handler function to be associated with the event.
cookie	an opaque pointer value to be provided unchanged as a parameter of the handler function call.
masked	indicates if the handle is masked initially.

Returns

An opaque pointer value in the ERR_PTR() format which uniquely identifies the event-handle. This value is to be provided as a parameter in any subsequent call to svec_evt_detach(), svec_evt_mask(), and svec_evt_unmask().

Definition at line 296 of file svec.c.

9.19.2.4 svec_evt_detach()

```
int svec_evt_detach (
            struct svec_event_handle * handle )
```

Detaches (i.e. dissociates) an event-handle from an event.

A detached event-handle does no longer has its associated handler called upon occurrence of the event. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

an errno value which indicates the operation success or failure:

- 0: success.
- a negative value: failure.

Definition at line 341 of file svec.c.

9.19.2.5 svec_evt_mask()

```
int svec_evt_mask (
            struct svec_event_handle * handle )
```

Masks (i.e. disables) an event-handle.

During the period of time an event-handle is masked, the associated handler is no longer called upon the event occurrence, the event is memorized as pending instead, until the event-handle is unmasked again. Masking/unmasking sequences can be nested. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

an errno value which indicates the operation success or failure:

- 0: success.
- a negative value: failure.

Definition at line 375 of file svec.c.

9.19.2.6 svec_evt_unmask()

```
int svec_evt_unmask (
            struct svec_event_handle * handle )
```

Unmasks (i.e. enables) an event-handle.

Enables the call of the handler associated with this event upon occurrence. If the event was pending for this event-handle at the time of unmasking, the associated handler is called. Masking/unmasking sequences can be nested. Can be called from any context (task or interrupt). In particular can be called from within an event handler function.

Parameters

handle	the opaque pointer value which uniquely identifies the event-handle as returned by svec_evt_attach().

Returns

an errno value which indicates the operation success or failure:

- success: 0.
- failure: a negative errno value.

Definition at line 413 of file svec.c.

9.19.2.7 svec_lookup()

```
struct svec* svec_lookup (
            const char * name )
```

Finds a Simple Virtual Event Controller device.

The svec-device is referenced to by its name as it appears in the device DT node "info" property. Multiple opens of the same device are allowed. Can be called from any context (task or interrupt).

Parameters

name	the name of the device.

Returns

An opaque pointer which uniquely identifies the svec-device instance. This value is to be provided as a parameter in any subsequent call to svec_evt_attach(), or NULL if none svec-device instance was found.

Definition at line 453 of file svec.c.

9.20 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vaudio.h File Reference

Provide virtual AUDIO driver interface.

```
#include <nk/nkern.h>
```

Data Structures

- struct NkEventOpen
- struct NkEventSetRate
- struct NkCtlElemInfo
- struct NkCtlElemValue
- struct NkEventMixer
- struct NkVaudioHwPcm
- struct NkVaudioHw
- struct NkVaudioCtrl
- struct NkVaudioMixer
- struct NkRingDesc
- struct vaudio_stream_shmem_t
- struct vaudio_shmem_t
- union NkCtlElemInfo.value
- struct NkCtlElemInfo.value.integer
- struct NkCtlElemInfo.value.enumerated
- union NkCtlElemValue.value
- struct NkCtlElemValue.value.integer
- struct NkCtlElemValue.value.enumerated
- struct vaudio_shmem_t.mixer

Macros

- #define NK_VAUDIO_SS_TYPE_INVAL 0
- #define NK_VAUDIO_ST_TYPE_INVAL 0
- #define NK_VAUDIO_MIXER_INFO 1
- #define NK_CTL_ELEM_TYPE_NONE 0
- #define NK_VAUDIO_MIXER_MAX 128
- #define HW_PCM_FMTBIT_S8 (1ULL << HW_PCM_FORMAT_S8)
- #define HW_PCM_RATE_5512 (1<<0) /* 5512Hz */
- #define HW_CAP_PCM 0x00000001
- #define NK_VAUDIO_RING_DESC_NB 64

Typedefs

- typedef void(* NkVaudioEventHandler) (void *stream, NkVaudioEvent event, void *params, void *cookie)

Enumerations

- enum NkVaudioEvent {
  NK_VAUDIO_STREAM_OPEN = 0x0,
  NK_VAUDIO_STREAM_CLOSE = 0x1,
  NK_VAUDIO_STREAM_START = 0x2,
  NK_VAUDIO_STREAM_STOP = 0x3,
  NK_VAUDIO_STREAM_SET_RATE = 0x4,
  NK_VAUDIO_STREAM_DATA = 0x5,
  NK_VAUDIO_STREAM_MIXER = 0x6 }
- enum vaudio_format_t
- enum vaudio_status_t

Functions

- bool vaudio_configured (NkOsId osid)
- NkVaudio ∗ vaudio_create (NkDevVlink ∗vlink, NkVaudioEventHandler hdl, void ∗cookie, const NkVaudioHw ∗hw_conf)
- int vaudio_start (NkVaudio ∗vaudio)
- void vaudio_destroy (NkVaudio ∗vaudio)
- int vaudio_ring_get (NkStream ∗stream, NkPhAddr ∗addr, nku32_f ∗size)
- void vaudio_ring_put (NkStream ∗stream, vaudio_status_t status)
- void vaudio_event_ack (NkVaudio ∗vaudio, NkStream ∗stream, NkVaudioEvent event, void ∗params, vaudio_status_t status)

9.20.1 Detailed Description

Provide virtual AUDIO driver interface.

9.20.2 Data Structure Documentation

9.20.2.1 struct NkEventOpen

NK_VAUDIO_STREAM_OPEN event parameters: session_type : playback or capture stream_type : PCM only

Definition at line 118 of file vaudio.h.

9.20.2.2 struct NkEventSetRate

NK_VAUDIO_STREAM_SET_RATE event parameters: channels : number of channels format : endianess and size of samples rate : rate in HZ period : period size in bytes periods : number of periods in the ring buffer dma_paddr: dma physical address of the ring buffer dma_vaddr: virtual address of the ring buffer This is not a shared memory object between backend and frontend.

Definition at line 163 of file vaudio.h.

9.20.2.3 struct NkCtlElemInfo

Definition at line 194 of file vaudio.h.

9.20.2.4 struct NkCtlElemValue

Definition at line 213 of file vaudio.h.

9.20.2.5 struct NkEventMixer

Definition at line 224 of file vaudio.h.

9.20.2.6 struct NkVaudioHwPcm

NkVaudioHwPcm : configuration for the PCM stream type. formats : PCM formats supported rates : sample rates supported rate_min : minimal rate in HZ rate_min : maximal rate in HZ channels_min : minimal number of channels channels_max : maximal number of channels

Definition at line 296 of file vaudio.h.

9.20.2.7 struct NkVaudioHw

NkVaudioHw : hardware configuration for the audio device. stream_cap : supported stream types pcm : PCM configuration

Definition at line 322 of file vaudio.h.

9.20.2.8 struct NkVaudioCtrl

Virtual audio control definition.

Definition at line 348 of file vaudio.h.

9.20.2.9 struct NkVaudioMixer

Definition at line 363 of file vaudio.h.

9.20.2.10 struct NkRingDesc

Definition at line 382 of file vaudio.h.

9.20.2.11 struct vaudio_stream_shmem_t

Definition at line 489 of file vaudio.h.

9.20.2.12 struct vaudio_shmem_t

Definition at line 501 of file vaudio.h.

9.20.2.13 union NkCtlElemInfo.value

Definition at line 198 of file vaudio.h.

9.20.2.14 struct NkCtlElemInfo.value.integer

Definition at line 199 of file vaudio.h.

9.20.2.15 struct NkCtlElemInfo.value.enumerated

Definition at line 204 of file vaudio.h.

9.20.2.16 union NkCtlElemValue.value

Definition at line 214 of file vaudio.h.

9.20.2.17 struct NkCtlElemValue.value.integer

Definition at line 215 of file vaudio.h.

9.20.2.18 struct NkCtlElemValue.value.enumerated

Definition at line 218 of file vaudio.h.

9.20.2.19 struct vaudio_shmem_t.mixer

Definition at line 503 of file vaudio.h.

9.20.3 Macro Definition Documentation

9.20.3.1 NK_VAUDIO_SS_TYPE_INVAL

```
#define NK_VAUDIO_SS_TYPE_INVAL 0
```

Values for the session_type field of the NkEventOpen event.

Definition at line 103 of file vaudio.h.

9.20.3.2 NK_VAUDIO_ST_TYPE_INVAL

```
#define NK_VAUDIO_ST_TYPE_INVAL 0
```

Values for the stream_type field of the NkEventOpen event.

Definition at line 110 of file vaudio.h.

9.20.3.3 NK_VAUDIO_MIXER_INFO

```
#define NK_VAUDIO_MIXER_INFO 1
```

Values for the mix_cmd field of the NkEventMixer event.

Definition at line 176 of file vaudio.h.

9.20.3.4 NK_CTL_ELEM_TYPE_NONE

```
#define NK_CTL_ELEM_TYPE_NONE 0
```

Values for the mix_info.type field of the NkEventMixer event.

Definition at line 183 of file vaudio.h.

9.20.3.5 NK_VAUDIO_MIXER_MAX

```
#define NK_VAUDIO_MIXER_MAX 128
```

NK_VAUDIO_STREAM_MIXER event parameters.

Definition at line 192 of file vaudio.h.

9.20.3.6 HW_PCM_FMTBIT_S8

```
#define HW_PCM_FMTBIT_S8 (1ULL << HW_PCM_FORMAT_S8)
```

Values for the formats field of the NkVaudioHwPcm configuration.

Definition at line 254 of file vaudio.h.

9.20.3.7 HW_PCM_RATE_5512

```
#define HW_PCM_RATE_5512 (1<<0) /* 5512Hz */
```

Values for the rates field of the NkVaudioHwPcm configuration.

Definition at line 273 of file vaudio.h.

9.20.3.8 HW_CAP_PCM

```
#define HW_CAP_PCM 0x00000001
```

Values for the stream_cap field of the NkVaudioHw configuration.

Definition at line 315 of file vaudio.h.

9.20.3.9 NK_VAUDIO_RING_DESC_NB

```
#define NK_VAUDIO_RING_DESC_NB 64
```

Ring of descriptors definition.

Definition at line 379 of file vaudio.h.

9.20.4 Typedef Documentation

9.20.4.1 NkVaudioEventHandler

```
typedef void(* NkVaudioEventHandler) (void *stream, NkVaudioEvent event, void *params, void
*cookie)
```

Call-back function prototype for virtual AUDIO events. The NkVaudioEventHandler is called by the virtual AUDIO interrupt handler on occurrence of an NkVaudioEvent event.

So an event handler should not use APIs which are not allowed by the underlying operating system within interrupt handlers.

stream : NkStream cookie. event : NkVaudioEvent event. params : pointer to event parameters. cookie : cookie given to vaudio_create().

Definition at line 246 of file vaudio.h.

9.20.5 Enumeration Type Documentation

9.20.5.1 NkVaudioEvent

```
enum NkVaudioEvent
```

Here are the virtual AUDIO events.

Enumerator

NK_VAUDIO_STREAM_OPEN	Indicate that remote site has opened the device.
NK_VAUDIO_STREAM_CLOSE	Indicate that remote site has closed the device.
NK_VAUDIO_STREAM_START	Start the audio stream.
NK_VAUDIO_STREAM_STOP	Stop the audio stream.
NK_VAUDIO_STREAM_SET_RATE	Set the rate of the audio stream.
NK_VAUDIO_STREAM_DATA	Indicate that a data buffer is available.
NK_VAUDIO_STREAM_MIXER	Get/set stream volume control.

Definition at line 64 of file vaudio.h.

9.20.5.2 vaudio_format_t

```
enum vaudio_format_t
```

Values for the format field of the NkEventSetRate event.

Definition at line 128 of file vaudio.h.

9.20.5.3 vaudio_status_t

```
enum vaudio_status_t
```

Error codes for the vaudio_ring_put and vaudio_event_ack status parameters.

Definition at line 332 of file vaudio.h.

9.20.6 Function Documentation

9.20.6.1 vaudio_configured()

```
bool vaudio_configured (
            NkOsId osid )
```

This routine checks if vaudio is configured for a given frontend osid.

Parameters

osid	: NkOsId of the frontend driver

Returns

In case of success this routine returns TRUE.

Definition at line 251 of file vaudio.c.

9.20.6.2 vaudio_create()

```
NkVaudio* vaudio_create (
            NkDevVlink * vlink,
            NkVaudioEventHandler hdl,
            void * cookie,
            const NkVaudioHw * hw_conf )
```

This routine creates a virtual audio device.

Parameters

vlink	: NK vlink descriptor
hdl	: event handler.
cookie	: client cookie passed back to the event handler.
hw_conf	: hardware AUDIO configuration.

Returns

In case of success this routine returns a handle to the created vaudio. Otherwise 0 is returned in case of failure.

Definition at line 257 of file vaudio.c.

9.20.6.3 vaudio_start()

```
int vaudio_start (
            NkVaudio * vaudio )
```

This routine starts a virtual audio device: attaches cross-interrupt handlers and initiates the handshake with peer.

Parameters

vaudio	: handle returned by vaudio_create().

Returns

In case of success this routine returns 1. Otherwise 0 is returned and vaudio_destroy() must be called to clean up.

Definition at line 313 of file vaudio.c.

9.20.6.4 vaudio_destroy()

```
void vaudio_destroy (
            NkVaudio * vaudio )
```

This routine destroys a virtual audio device.

Definition at line 241 of file vaudio.c.

9.20.6.5 vaudio_ring_get()

```
int vaudio_ring_get (
            NkStream * stream,
            NkPhAddr * addr,
            nku32_f * size )
```

Get a data buffer from the virtual AUDIO device. If the opened stream is in playback mode, the buffer contains data to be played by the audio device. If the opened stream is in capture mode, the buffer will be filled by the audio device.

Parameters

stream	: the stream handle.
addr	: pointer filled with the physical address of the buffer.
size	: pointer filled with the size of the buffer.

Returns

1 in case of success.
0 when no buffer is available.

Definition at line 347 of file vaudio.c.

9.20.6.6 vaudio_ring_put()

```
void vaudio_ring_put (
            NkStream * stream,
            vaudio_status_t status )
```

Put a data buffer to the virtual AUDIO device. If the opened stream is in capture mode, the buffer contains data captured by the audio device. If the opened stream is in playback mode, the buffer has been played by the audio device.

Parameters

stream	: the stream handle.
status	: status of the buffer: NK_VAUDIO_STATUS_OK operation has succeeded NK_VAUDIO_STATUS_ERROR operation has failed

Definition at line 366 of file vaudio.c.

9.20.6.7 vaudio_event_ack()

```
void vaudio_event_ack (
            NkVaudio * vaudio,
            NkStream * stream,
            NkVaudioEvent event,
            void * params,
            vaudio_status_t status )
```

Acknowledge an event to the virtual AUDIO device.

Parameters

vaudio	: the virtual audio device handle.
stream	: the stream handle.
event	: the event to be acknowledged.
params	: pointer to the event parameters.
status	: status of the event: NK_VAUDIO_STATUS_OK operation has succeeded NK_VAUDIO_STATUS_ERROR operation has failed

Definition at line 382 of file vaudio.c.

9.21 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_client.h File Reference

```
#include <linux/kref.h>
#include <linux/list.h>
#include <linux/mutex.h>
#include "vdmaheap_dev.h"
```

Data Structures

- struct vdmaheap_grace

  Grace delay property data structure used in the purpose of caching imports. More...
- struct vdmaheap_client_props

  Client's property data structure. More...
- struct vdmaheap_client

  client data structure. More...

Functions

- struct vdmaheap_client ∗ vdmaheap_client_create (struct vdmaheap_device ∗vdev)

  creates a client.
- void vdmaheap_client_destroy (struct vdmaheap_client ∗client)

  destroys a client.

9.21.1 Detailed Description

vdmaheap_client.h - virtual DMAHEAP driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.lizzi@harman.com)

9.21.2 Data Structure Documentation

9.21.2.1 struct vdmaheap_grace

Grace delay property data structure used in the purpose of caching imports.

This data structure contains the parameters needed by vDMAHEAP to perform import caching. In particular, it contains the grace delay value requested for any import request issued by this client, and the maximal grace delay accepted by this client for any received import request. Both requested and accepted grace delays are provided in Jiffies. When an import is performed, the grace delay actually associated with the buffer results from the negotiation between the issuer and the receiver clients according to the rule actual = MIN(requested, accepted). By default a client is assigned requested = 0, accepted = -1, meaning it does not use the import cache for the imports requests it issues, but accepts any grace delay value for any import request it receives.

Definition at line 47 of file vdmaheap_client.h.

Data Fields

unsigned long	requested	grace delay requested for any import request issued by this client, in Jiffies
unsigned long	accepted	maximal grace delay accepted by this client for any received import request, in Jiffies

9.21.2.2 struct vdmaheap_client_props

Client's property data structure.

This data structures describes the properties attached to a vDMAHEAP client. These properties can be set and gotten through the vdmaheap_kapi_set_client_props() and vdmaheap_kapi_get_client_props() interface functions.

Definition at line 62 of file vdmaheap_client.h.

Data Fields

bool	strict_import	if true, the vDMAHEAP enforces the strict import semantic for this client
struct vdmaheap_grace	grace	grace property to be used for any import request issued or received by this client in the purpose of caching imports

9.21.2.3 struct vdmaheap_client

client data structure.

any vdmaheap operation on a dmabuf is achieved through a client. All clients are referenced in a list of the vdmaheap_device they belong to. Each client references in the list cred_list the vdmaheap_cred objects it retrieved upon exports. The client is the container of some properties like trusted, strict_import, grace.

Definition at line 79 of file vdmaheap_client.h.

Data Fields

struct kref	kref	reference counter held by vdmaheap_buffer and vdmaheap_import
struct list_head	list_node	node for the vdmaheap_device list
struct vdmaheap_device ∗	dev	pointer to the vdmaheap_device the client belongs to
struct mutex	cred_list_lock	protects the cred_list against concurrent accesses
unsigned int	version	
struct list_head	cred_list	list of the credentials created by this client
bool	trusted	trusted attribute
bool	strict_import	strict-import attribute
struct vdmaheap_grace	grace	requested-grace and accepted-grace attributes
char	name[20]	client name

9.21.3 Function Documentation

9.21.3.1 vdmaheap_client_create()

```
struct vdmaheap_client* vdmaheap_client_create (
            struct vdmaheap_device * vdev )
```

creates a client.

Parameters

vdev	a pointer on the vdmaheap_device data structure the client is to be created in.

Returns

- NULL: the client failed to be created.
- valid pointer: the address of the created vdmaheap_client data structure.

Definition at line 131 of file vdmaheap_client.c.

9.21.3.2 vdmaheap_client_destroy()

```
void vdmaheap_client_destroy (
            struct vdmaheap_client * client )
```

destroys a client.

Parameters

client	a pointer of the client to be destroyed.

Definition at line 154 of file vdmaheap_client.c.

9.22 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_export.h File Reference

```
#include <linux/kref.h>
#include <linux/list.h>
#include <linux/dma-buf.h>
#include <linux/uuid.h>
#include "vdmaheap_dev.h"
```

Data Structures

- struct vdmaheap_cred

    credential for an exported dmabuf More...

Functions

- struct dma_buf ∗ vdmaheap_dmabuf_local_import (struct vdmaheap_client ∗vclient, vdmaheap_buffer_gid_t gid, const uuid_t ∗cred)

    imports a local dmabuf.

- int vdmaheap_dmabuf_export (struct vdmaheap_client ∗vclient, struct dma_buf ∗dmabuf, vdmaheap_buffer_gid_t ∗gidp, uuid_t ∗cred, u32 ∗size)

  exports a dmabuf.
- int vdmaheap_dmabuf_unexport (struct vdmaheap_client ∗vclient, vdmaheap_buffer_gid_t gid, const uuid_t ∗cred)

  unexports a dmabuf.
- int vdmaheap_dev_ioc_export (struct vdmaheap_client ∗vclient, struct vdmaheap_export_data ∗data)

  exports a dmabuf from user-space.
- int vdmaheap_dev_ioc_unexport (struct vdmaheap_client ∗vclient, struct vdmaheap_unexport_data ∗data)

  unexports a dmabuf from user-space.
- int vdmaheap_vbuf_req_release (struct vdmaheap_be ∗vbe, struct vdmaheap_buffer ∗vbuf)

  requests the exporter to release a dmabuf.
- struct vdmaheap_buffer ∗ vdmaheap_vbuf_req_import (struct vdmaheap_be ∗vbe, vdmaheap_buffer_gid_t gid, const uuid_t ∗cred, u32 ∗sg_size)

  requests the exporter to import a dmabuf.
- int vdmaheap_vbuf_rsp_import (struct vdmaheap_be ∗vbe, struct vdmaheap_buffer ∗vbuf, nku32_f gflags, struct vdmaheap_rsp_import_mult_descr ∗descr)

  completes or cancel a dmabuf import.

9.22.1 Detailed Description

vdmaheap_export.h - virtual DMAHEAP driver

Copyright (c) 1999-2022 HARMAN. All Rights Reserved.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License Version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

You should have received a copy of the GNU General Public License Version 2 along with this program. If not, see http://www.gnu.org/licenses/.

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.↩ lizzi@harman.com)

9.22.2 Data Structure Documentation

9.22.2.1 struct vdmaheap_cred

credential for an exported dmabuf

Definition at line 45 of file vdmaheap_export.h.

Data Fields

struct list_head	client_list_node	node for the vdmaheap_client list
struct list_head	vbuf_list_node	node for the vdmaheap_buffer list
uuid_t	cred	the credential at properly speaking
unsigned int	flags	
struct kref	kref	reference counter held by the vdmaheap_client
struct vdmaheap_buffer ∗	vbuf	points to the vdmaheap_buffer this vdmaheap_cred belongs to
struct vdmaheap_client ∗	client	points to the vdmaheap_client this vdmaheap_cred belongs to

9.22.3 Function Documentation

9.22.3.1 vdmaheap_dmabuf_local_import()

```
struct dma_buf* vdmaheap_dmabuf_local_import (
            struct vdmaheap_client * vclient,
            vdmaheap_buffer_gid_t gid,
            const uuid_t * cred )
```

imports a local dmabuf.

operates with the same input/output parameter than the remote variant. The difference lies in the fact that the returned imported dmabuf is the same object that the exported one.

Parameters

vclient	the client which imports the dmabufs.
gid	the GID of the dmabuf to be imported
cred	a pointer on the credentials,

Returns

- NULL: the dmabuf failed to be imported.
- valid pointer: the address of the imported dmabuf.

Definition at line 722 of file vdmaheap_export.c.

9.22.3.2 vdmaheap_dmabuf_export()

```
int vdmaheap_dmabuf_export (
            struct vdmaheap_client * vclient,
            struct dma_buf * dmabuf,
            vdmaheap_buffer_gid_t * gidp,
            uuid_t * cred,
            u32 * size )
```

exports a dmabuf.

takes a dmabuf address as input, and produces a couple (GID, credential) as output. The GID is associated with the dmabuf, whereas the credential is associated with the export operation. If the same dmabuf is exported several times, the same GID is produced each time, a different credential is produced each time. This function is the core export function used by kapi, vbb and uapi to provide the export service.

Parameters

vclient	the client which exports the dmabuf.
dmabuf	a pointer to the dmabuf to be exported.
gidp	a pointer to the GID of the exported dmabuf.
cred	a pointer to the credentials of the exported dmabuf.
size	a pointer to the size of the exported dmabuf, can be NULL.

Returns

- 0: the dmabuf could be successfully exported.
- -ENOMEM: the vdmaheap_export object could not be allocated, or could not be added to the vbuf_tree.

Definition at line 995 of file vdmaheap_export.c.

9.22.3.3 vdmaheap_dmabuf_unexport()

```
int vdmaheap_dmabuf_unexport (
            struct vdmaheap_client * vclient,
            vdmaheap_buffer_gid_t gid,
            const uuid_t * cred )
```

unexports a dmabuf.

takes as input a couple (GID, credential) produced by a prior export operation. Once unexported the couple (GID, credential) can no longer be imported. It remains alive for all importers that had imported it before the unexport. This function is the core unexport function used by kapi, vbb and uapi to provide the unexport service.

Parameters

vclient	the client which unexports the dmabuf.
gid	the GID of the dmabuf to be unexported.
cred	the credentials of the dmabuf to be unexported, can be NULL.

Returns

- 0: the dmabuf could be successfully exported.
- -ENOENT: the dmabuf referenced by the GID is not (no longer) exported.
- -EACCES: the credentials are incorrect.

Definition at line 1108 of file vdmaheap_export.c.

9.22.3.4 vdmaheap_dev_ioc_export()

```
int vdmaheap_dev_ioc_export (
            struct vdmaheap_client * vclient,
            struct vdmaheap_export_data * data )
```

exports a dmabuf from user-space.

takes a file descriptor referencing a dmabuf as input, and produces a couple (GID, credential) as output. This function is used by the ioctl function.

Parameters

vclient	the client which exports the dmabuf.
data	the export data structure exposed to the user-space. It contains a file descriptor as intput, a GID and a credential as output.

Returns

- 0: the dmabuf could be successfully exported.
- negative value: indicates the cause of the failure.

Definition at line 1042 of file vdmaheap_export.c.

9.22.3.5 vdmaheap_dev_ioc_unexport()

```
int vdmaheap_dev_ioc_unexport (
            struct vdmaheap_client * vclient,
            struct vdmaheap_unexport_data * data )
```

unexports a dmabuf from user-space.

takes as input a couple (GID, credential) produced by a prior export operation. This function is used by the ioctl function.

Parameters

vclient	the client which unexports the dmabuf.
data	the unexport data structure exposed to the user-space. It contains a GID and a credential as input.

Returns

- 0: the dmabuf could be successfully unexported.
- negative value: indicates the cause of the failure.

Definition at line 615 of file vdmaheap_export.c.

9.22.3.6 vdmaheap_vbuf_req_release()

```
int vdmaheap_vbuf_req_release (
            struct vdmaheap_be * vbe,
            struct vdmaheap_buffer * vbuf )
```

requests the exporter to release a dmabuf.

a release request was received by the vrpc layer.

Parameters

vbe	the local vrpc link endpoint the request was received through.
vbuf	a pointer to the vdmaheap_buffer representing the dmabuf to be released on the exporter's side.

Returns

- 0: the dmabuf was successfully released.
- EACCES: the dmabuf could not be denied the access.

Definition at line 1208 of file vdmaheap_export.c.

9.22.3.7 vdmaheap_vbuf_req_import()

```
struct vdmaheap_buffer* vdmaheap_vbuf_req_import (
            struct vdmaheap_be * vbe,
            vdmaheap_buffer_gid_t gid,
            const uuid_t * cred,
            u32 * sg_size )
```

requests the exporter to import a dmabuf.

an import request was received by the vrpc layer.

Parameters

vbe	the local vrpc link endpoint the request was received through.
gid	the Global Identifier of the dmabuf to be imported.
cred	a pointer to a uuid_t data structure containig the credentials of the dmabuf, can be NULL is the importer client is trusted.
sg_size	a pointer to a variable containing the size of the dmabuf sg_table, can be NULL if the layout is not needed.

Returns

an ERR_PTR pointer:

- a valid pointer to the vdmaheap_buffer representing the imported dmabuf on the exporter's side, if the import can be carried on with vdmaheap_vbuf_rsp_import().
- an errno value indicating the cause of the failure otherwise.
  - -ENOENT: the dmabuf does not (longer) exist.
  - -EACCES: the credential are incorrect.

Definition at line 1302 of file vdmaheap_export.c.

9.22.3.8 vdmaheap_vbuf_rsp_import()

```
int vdmaheap_vbuf_rsp_import (
            struct vdmaheap_be * vbe,
            struct vdmaheap_buffer * vbuf,
            nku32_f gflags,
            struct vdmaheap_rsp_import_mult_descr * descr )
```

completes or cancel a dmabuf import.

an import request was received by the vrpc layer.

Parameters

vbe	the local vrpc link endpoint the request was received through.
vbuf	a pointer to the vdmaheap_buffer being imported.
gflags	the 'memory-granting-flags' value to be used.
descr	a pointer to a descr data structure to be filled (layout; size), if the import is carried-on, or NULL if the import is to be cancelled.

Returns

- 0: the dmabuf could be successfully imported.
- EACCES: the access could not be granted.

Definition at line 1339 of file vdmaheap_export.c.

9.23 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_import.h File Reference

```
#include <linux/dma-buf.h>
#include <linux/uuid.h>
#include "vrpc.h"
#include "vdmaheap_dev.h"
```

Functions

- int vdmaheap_dev_ioc_import (struct vdmaheap_client *vclient, struct vdmaheap_import_data *data)

  imports a dmabuf from user-space.
- int vdmaheap_dmabuf_import (struct vdmaheap_client *vclient, unsigned int n, vdmaheap_buffer_gid_t gids[ ], const uuid_t *creds[ ], struct dma_buf *dmabufs[ ])

  imports a set dmabufs.

9.23.1 Detailed Description

vdmaheap_import.h - virtual DMAHEAP driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.↩
lizzi@harman.com)

9.23.2 Function Documentation

9.23.2.1 vdmaheap_dev_ioc_import()

```
int vdmaheap_dev_ioc_import (
            struct vdmaheap_client * vclient,
            struct vdmaheap_import_data * data )
```

imports a dmabuf from user-space.

takes as input a couple (GID, credential) produced by a prior export operations, and produces as output a file descriptor referencing the dmabuf build locally. This function is used by the ioctl function.

Parameters

vclient	the client which imports the dmabufs.
data	the import data structure exposed to the user-space. It contains a GID, a credential as input, a file descriptor as output.

Returns

- 0: the dmabuf could be successfully imported.
- negative value: indicates the cause of the failure.

Definition at line 2013 of file vdmaheap_import.c.

9.23.2.2 vdmaheap_dmabuf_import()

```
int vdmaheap_dmabuf_import (
            struct vdmaheap_client * vclient,
            unsigned int n,
            vdmaheap_buffer_gid_t gids[],
            const uuid_t * creds[],
            struct dma_buf * dmabufs[] )
```

imports a set dmabufs.

takes as input an array of couples (GID, credential) produced by prior export operations, produces as output, an array of pointers on dmabufs in the ERR_PTR format. Valid dmabuf point to dmabuf objects which have been rebuild locally. Invalid dmabufs indicate the cause of the failure. This function is the core import function used by kapi, vbb and uapi to provide the import service.

Parameters

vclient	the client which imports the dmabufs.
n	the number of dmabufs to be imported.
gids	an array of n GIDs.
creds	an array of n credentials, can be NULL.
dmabufs	an array of n pointers to dmabufs,

Returns

- 0: all the dmabuf could be successfully imported.
- -ETIME:
- -EPIPE:

Definition at line 1893 of file vdmaheap_import.c.

9.24 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vbb.h File Reference

Functions

- struct __vbb_context ∗ vbb_context_create (const char ∗name)

    creates a UBM dedicated kernel-space client.
- int vbb_context_release (struct __vbb_context ∗ctx)

    release a UBM dedicated kernel-space client.
- int vbb_destroy_buf_id (struct __vbb_context ∗ctx, unsigned int buf_id)

    unexports a dmabuf.
- unsigned int vbb_create_buf_id (struct __vbb_context ∗ctx, struct dma_buf ∗dmabuf)

    exports a dmabuf.

9.24.1 Detailed Description

vdmaheap_vbb.h - virtual DMAHEAP driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.lizzi@harman.com)

9.24.2 Function Documentation

9.24.2.1 vbb_context_create()

```
struct __vbb_context* vbb_context_create (
            const char * name )
```

creates a UBM dedicated kernel-space client.

as any regular kernel-space clients, it has trusted and does not use the strict_import semantic.

Parameters

name	the client name.

Returns

the created client address in the ERR_PTR format.

Definition at line 39 of file vdmaheap_vbb.c.

9.24.2.2 vbb_context_release()

```
int vbb_context_release (
            struct __vbb_context * ctx )
```

release a UBM dedicated kernel-space client.

Parameters

ctx	the address of the client to be released.

Returns

- 0: the client was successfully released.
- negative value: indicates the cause of the failure.

Definition at line 58 of file vdmaheap_vbb.c.

9.24.2.3 vbb_destroy_buf_id()

```
int vbb_destroy_buf_id (
            struct __vbb_context * ctx,
            unsigned int buf_id )
```

unexports a dmabuf.

unlike its corresponding regular kernel-API vdmaheap_kapi_unexport() function, it does not take credentials as input parameter.

Parameters

ctx	the address of the client the unexport is issued through.
buf_id	the GID of the dmabuf to be unexported.

Returns

- 0: the dmabuf was successfully unexported.
- negative value: indicates the cause of the failure.

Definition at line 88 of file vdmaheap_vbb.c.

9.24.2.4 vbb_create_buf_id()

```
unsigned int vbb_create_buf_id (
            struct __vbb_context * ctx,
            struct dma_buf * dmabuf )
```

exports a dmabuf.

unlike its corresponding regular kernel-API vdmaheap_kapi_export(), function, it does not generate credentials for the exported dmabuf.

Parameters

ctx	the address of the client the export is issued through.
dmabuf	the address of the dmabuf to be exported.

Returns

- 0: the dmabuf failed to be exported,
- positive value: the GID assigned to the exported dmabuf.

Definition at line 70 of file vdmaheap_vbb.c.

9.25 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vdmaheap_vrpc.h File Reference

```
#include <linux/radix-tree.h>
#include <linux/list.h>
#include <linux/mutex.h>
#include <linux/workqueue.h>
#include <linux/uuid.h>
#include <nk/nkern.h>
#include <vlx/pv-version-interface.h>
#include "vrpc.h"
#include "vdmaheap_dev.h"
```

Data Structures

- struct vdmaheap_rsp_import_mult_descr
- struct vdmaheap_vrpc_stats
- struct vdmaheap_be

  vdmaheap back-end data structure. More...
- struct vdmaheap_fe

Functions

- int vdmaheap_fe_req_release (struct vdmaheap_fe ∗vfe, vdmaheap_buffer_gid_t gid, struct vdmaheap_import ∗vimp)

  requests the vrpc layer to release a dmabuf.
- int vdmaheap_fe_req_import (struct vdmaheap_fe ∗vfe, nku32_f gflags, unsigned int ∗nbufs, struct vdmaheap_import ∗vimps[ ], const uuid_t ∗creds[ ], int ress[ ], unsigned int flags[ ], vrpc_size_t ∗size)

  requests the vrpc layer to import a set of dmabufs.
- int vdmaheap_fe_req_release_flush (struct vdmaheap_fe ∗vfe)

  requests the vrpc layer to flush all the pending release requests.

9.25.1 Detailed Description

vdmaheap_vrpc.h - virtual DMAHEAP driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.↵
lizzi@harman.com)

9.25.2 Data Structure Documentation

9.25.2.1 struct vdmaheap_rsp_import_mult_descr

Definition at line 44 of file vdmaheap_vrpc.h.

9.25.2.2 struct vdmaheap_vrpc_stats

Definition at line 60 of file vdmaheap_vrpc.h.

9.25.2.3 struct vdmaheap_be

vdmaheap back-end data structure.

represents the exporter's side vdmaheap link endpoint. receives requests messages from its peer front-end.

Definition at line 72 of file vdmaheap_vrpc.h.

Data Fields

struct vdmaheap_device *	dev	points to the vdmaheap_device this vdmaheap_be belongs to
struct mutex	lock	protects this data structure against concurrent accesses
struct list_head	list_node	node for the vdmaheap_device list of back-ends
struct vrpc_t *	vrpc	points to the vrpc back-end underneath
void *	vrpc_data	pointer to the communication memory (i.e. pmem)
vrpc_size_t	vrpc_maxsize	pmem size in bytes
int	vrpc_open	indicates if the vrpc link is open
void *	safe_data	points to a safe memory area where import request is copied
vrpc_size_t	safe_size	size of the safe memory area
unsigned int	vrpc_peer_id	vm identifier of the peer vdmaheap_fe entity
struct radix_tree_root	export_tree	references by their GIDs all the buffers exported by this vdmaheap_be
struct work_struct	peer_off_work	work scheduled when the vlink goes down
struct vdmaheap_vrpc_stats	stats	statistics associated with this vdmaheap_be
struct pv_driver_header_v1_t	pv_ver	version of this vdmaheap_be

9.25.2.4 struct vdmaheap_fe

Definition at line 105 of file vdmaheap_vrpc.h.

Data Fields

struct vdmaheap_device *	dev	points to the vdmaheap_device this vdmaheap_be belongs to
struct mutex	lock	protects this data structure and the pmem against concurrent accesses
struct list_head	list_node	node for the vdmaheap_device list of front-ends
struct vrpc_t *	vrpc	points to the vrpc front-end underneath
void *	vrpc_data	pointer to the communication memory (i.e. pmem)
vrpc_size_t	vrpc_maxsize	pmem size in bytes
unsigned int	vrpc_peer_id	vm identifier of the peer vdmaheap_be entity
int	vrpc_open	
int	link_open	
bool	pending_release	
struct radix_tree_root	import_tree	
struct work_struct	peer_off_work	work scheduled when the vlink goes down
struct delayed_work	flush_release_dwork	work scheduled when a timeout expires after a release without import
struct vdmaheap_vrpc_stats	stats	statistics associated with this vdmaheap_fe
struct pv_driver_header_v1_t	pv_ver	version of this vdmaheap_fee

9.25.3 Function Documentation

9.25.3.1 vdmaheap_fe_req_release()

```
int vdmaheap_fe_req_release (
            struct vdmaheap_fe * vfe,
            vdmaheap_buffer_gid_t gid,
            struct vdmaheap_import * vimp )
```

requests the vrpc layer to release a dmabuf.

dmabuf release can be achieved in two ways:

- immediate release: a release request is immediately transmitted to the peer entity,
- deferred release: the release succeeds immediately and locally, the release request will transmitted to the peer entity later on. When this method is used, all the pending release requests must be transmitted to the peer entity before the next import request, in order to preserve import/release causality. This is done through vdmaheap_fe_req_release_flush(). The way release are performed is controlled through the internal compilation flag: VDMAHEAP_DEFERRED_RELEASE_ENABLED.

Parameters

vfe	the local vrpc link endpoint, the request is to be sent through.
gid	the buffer Global Identifier.
vimp	the data structure which represents the dmabuf on the importer's side.

Returns

- 0: if the request was successfully received, processed and answered by the peer entity.
- -ETIME: the request was not answered by the peer entity.

Definition at line 1028 of file vdmaheap_vrpc.c.

9.25.3.2 vdmaheap_fe_req_import()

```
int vdmaheap_fe_req_import (
            struct vdmaheap_fe * vfe,
            nku32_f gflags,
            unsigned int * nbufs,
            struct vdmaheap_import * vimps[ ],
            const uuid_t * creds[ ],
            int ress[ ],
            unsigned int flags[ ],
            vrpc_size_t * size )
```

requests the vrpc layer to import a set of dmabufs.

the vrpc layer sends an import request to the peer entity, containing the GIDs and credentials for each of the imported dmabuf.

Parameters

vfe	the local vrpc link endpoint, the request is to be sent through.
gflags	the 'memory-granting-flags' value to be used.
nbufs	the number of dmabuf to be imported.
vimps	an array of nbufs pointers to vdmaheap_import data structures to be filled after the import (e.g. the dmabuf layout, size, ...).
creds	an array of nbufs pointers to uuid_t data structures containing the dmabuf credentials, can be NULL if the importer client is trusted.
ress	an array of nbufs integers indicating the result of the dmabuf import in the errno format. A zero value indicates that the dmabuf was successfully imported, a negative value indicates the cause of the failure.
flags	an array of nbufs flags containing indications on how the import is to be performed. The only available flag indicates whether the dmabuf memory layout is needed.
size	the size of the import response message. This is only aimed at maintaining statistics.

Returns

- 0: if the request was successfully received, processed and answered by the peer entity.
- -EPIPE: the link with the peer entity is broken. When this happens, all imported dmabuf already imported must be closed.
- -ETIME: the request was not answered by the peer entity.

Definition at line 1125 of file vdmaheap_vrpc.c.

9.25.3.3 vdmaheap_fe_req_release_flush()

```
int vdmaheap_fe_req_release_flush (
            struct vdmaheap_fe * vfe )
```

requests the vrpc layer to flush all the pending release requests.

a sequence of release requests is sent to the peer entity (one for each pending request.

Parameters

vfe	the local vrpc link endpoint, the request is to be sent through.

Returns

- 0: all the pending release requests were successfully sent, processed and answered by the peer entity.
- -EPIPE: one pending release request at least could not be successfully sent due to fact that the vrpc link is broken. When this happens, the link is reported as broken, all the dmabuf already imported on this link are invalidated, and must be closed by their owners.

Definition at line 1280 of file vdmaheap_vrpc.c.

9.26 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vfence2_kapi.h File Reference

virtual DMA fence v2 (vFence2) driver - kernel mode API

```
#include <vlx/vfence2/uapi/vfence2_uapi.h>
#include <linux/types.h>
```

Functions

- int vfence2_create_client (const char ∗name, struct vfence2_client ∗∗vfcl) __must_check

    Create a vfence client.
- int vfence2_destroy_client (struct vfence2_client ∗vfcl)

    Destroy a vfence client.

- int vfence2_connect (struct vfence2_client ∗vfcl, uint32_t vmid, struct vfence2_connection ∗∗vfco) __must↩ _check

  Connect to a peer VM.
- int vfence2_disconnect (struct vfence2_connection ∗vfco)

  Disconnect a peer VM, previous connected with vfence2_connect()
- int vfence2_export_fence (struct vfence2_connection ∗vfco, int fd, vfence2_id_t ∗gid, bool can_block) __↩ must_check

  Export a DMA fence.
- int vfence2_unexport_fence (struct vfence2_connection ∗vfco, vfence2_id_t gid)

  Unexport a DMA fence.
- int vfence2_import_fence (struct vfence2_connection ∗vfco, vfence2_id_t gid, int ∗fd) __must_check

  Import a DMA fence.

9.26.1 Detailed Description

virtual DMA fence v2 (vFence2) driver - kernel mode API

This file defines the interface exposed by the vFence2 driver to kernel mode clients for the purpose of sharing DMA fences between virtual machines.

9.27 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_client.h File Reference

```
#include <linux/kref.h>
#include <linux/list.h>
#include <linux/mutex.h>
#include "vion_dev.h"
```

Data Structures

- struct vion_grace

  Grace delay property data structure used in the purpose of caching imports. More...
- struct vion_client_props

  Client's property data structure. More...
- struct vion_client

  client data structure. More...

Functions

- struct vion_client ∗ vion_client_create (struct vion_device ∗vdev)

  creates a client.
- void vion_client_destroy (struct vion_client ∗client)

  destroys a client.

9.27.1 Detailed Description

vion_client.h - virtual ION driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.lizzi@harman.com)

9.27.2 Data Structure Documentation

9.27.2.1 struct vion_grace

Grace delay property data structure used in the purpose of caching imports.

This data structure contains the parameters needed by vION to perform import caching. In particular, it contains the grace delay value requested for any import request issued by this client, and the maximal grace delay accepted by this client for any received import request. Both requested and accepted grace delays are provided in Jiffies. When an import is performed, the grace delay actually associated with the buffer results from the negotiation between the issuer and the receiver clients according to the rule actual = MIN(requested, accepted). By default a client is assigned requested = 0, accepted = -1, meaning it does not use the import cache for the imports requests it issues, but accepts any grace delay value for any import request it receives.

Definition at line 47 of file vion_client.h.

Data Fields

unsigned long	requested	grace delay requested for any import request issued by this client, in Jiffies
unsigned long	accepted	maximal grace delay accepted by this client for any received import request, in Jiffies

9.27.2.2 struct vion_client_props

Client's property data structure.

This data structures describes the properties attached to a vION client. These properties can be set and gotten through the vion_kapi_set_client_props() and vion_kapi_get_client_props() interface functions.

Definition at line 62 of file vion_client.h.

Data Fields

bool	strict_import	if true, the vION enforces the strict import semantic for this client
struct vion_grace	grace	grace property to be used for any import request issued or received by this client in the purpose of caching imports

9.27.2.3 struct vion_client

client data structure.

any vion operation on a dmabuf is achieved through a client. All clients are referenced in a list of the vion_device they belong to. Each client references in the list cred_list the vion_cred objects it retrieved upon exports. The client is the container of some properties like trusted, strict_import, grace.

Definition at line 79 of file vion_client.h.

Data Fields

struct kref	kref	reference counter held by vion_buffer and vion_import
struct list_head	list_node	node for the vion_device list
struct vion_device *	dev	pointer to the vion_device the client belongs to
struct mutex	cred_list_lock	protects the cred_list against concurrent accesses
unsigned int	version	
struct list_head	cred_list	list of the credentials created by this client
bool	trusted	trusted attribute
bool	strict_import	strict-import attribute
struct vion_grace	grace	requested-grace and accepted-grace attributes
char	name[20]	client name

9.27.3 Function Documentation

9.27.3.1 vion_client_create()

```
struct vion_client* vion_client_create (
            struct vion_device * vdev )
```

creates a client.

Parameters

vdev	a pointer on the vion_device data structure the client is to be created in.

Returns

- NULL: the client failed to be created.
- valid pointer: the address of the created vion_client data structure.

Definition at line 131 of file vion_client.c.

9.27.3.2 vion_client_destroy()

```
void vion_client_destroy (
            struct vion_client * client )
```

destroys a client.

Parameters

client	a pointer of the client to be destroyed.

Definition at line 154 of file vion_client.c.

9.28 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_export.h File Reference

```
#include <linux/kref.h>
#include <linux/list.h>
#include <linux/dma-buf.h>
#include <linux/uuid.h>
#include "vion_dev.h"
```

Data Structures

- struct vion_cred

  credential for an exported dmabuf More...

Functions

- struct dma_buf ∗ vion_dmabuf_local_import (struct vion_client ∗vclient, vion_buffer_gid_t gid, const uuid_t ∗cred)

  imports a local dmabuf.
- int vion_dmabuf_export (struct vion_client ∗vclient, struct dma_buf ∗dmabuf, vion_buffer_gid_t ∗gidp, uuid_t ∗cred, u32 ∗size)

  exports a dmabuf.
- int vion_dmabuf_unexport (struct vion_client ∗vclient, vion_buffer_gid_t gid, const uuid_t ∗cred)

  unexports a dmabuf.
- int vion_dev_ioc_export (struct vion_client ∗vclient, struct vion_export_data ∗data)

  exports a dmabuf from user-space.
- int vion_dev_ioc_unexport (struct vion_client ∗vclient, struct vion_unexport_data ∗data)

  unexports a dmabuf from user-space.
- int vion_vbuf_req_release (struct vion_be ∗vbe, struct vion_buffer ∗vbuf)

  requests the exporter to release a dmabuf.
- struct vion_buffer ∗ vion_vbuf_req_import (struct vion_be ∗vbe, vion_buffer_gid_t gid, const uuid_t ∗cred, u32 ∗sg_size)

  requests the exporter to import a dmabuf.
- int vion_vbuf_rsp_import (struct vion_be ∗vbe, struct vion_buffer ∗vbuf, nku32_f gflags, struct vion_rsp_import_mult_descr ∗descr)

  completes or cancel a dmabuf import.

9.28.1 Detailed Description

vion_export.h - virtual ION driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.↩
lizzi@harman.com)

9.28.2 Data Structure Documentation

9.28.2.1 struct vion_cred

credential for an exported dmabuf

Definition at line 45 of file vion_export.h.

Data Fields

struct list_head	client_list_node	node for the vion_client list
struct list_head	vbuf_list_node	node for the vion_buffer list
uuid_t	cred	the credential at properly speaking
unsigned int	flags	
struct kref	kref	reference counter held by the vion_client
struct vion_buffer ∗	vbuf	points to the vion_buffer this vion_cred belongs to
struct vion_client ∗	client	points to the vion_client this vion_cred belongs to

9.28.3 Function Documentation

9.28.3.1 vion_dmabuf_local_import()

```
struct dma_buf* vion_dmabuf_local_import (
            struct vion_client * vclient,
            vion_buffer_gid_t gid,
            const uuid_t * cred )
```

imports a local dmabuf.

operates with the same input/output parameter than the remote variant. The difference lies in the fact that the returned imported dmabuf is the same object that the exported one.

Parameters

vclient	the client which imports the dmabufs.
gid	the GID of the dmabuf to be imported
cred	a pointer on the credentials,

Returns

- NULL: the dmabuf failed to be imported.
- valid pointer: the address of the imported dmabuf.

Definition at line 726 of file vion_export.c.

9.28.3.2 vion_dmabuf_export()

```
int vion_dmabuf_export (
            struct vion_client * vclient,
            struct dma_buf * dmabuf,
            vion_buffer_gid_t * gidp,
            uuid_t * cred,
            u32 * size )
```

exports a dmabuf.

takes a dmabuf address as input, and produces a couple (GID, credential) as output. The GID is associated with the dmabuf, whereas the credential is associated with the export operation. If the same dmabuf is exported several times, the same GID is produced each time, a different credential is produced each time. This function is the core export function used by kapi, vbb and uapi to provide the export service.

Parameters

vclient	the client which exports the dmabuf.
dmabuf	a pointer to the dmabuf to be exported.
gidp	a pointer to the GID of the exported dmabuf.
cred	a pointer to the credentials of the exported dmabuf.
size	a pointer to the size of the exported dmabuf, can be NULL.

Returns

- 0: the dmabuf could be successfully exported.
- -ENOMEM: the vion_export object could not be allocated, or could not be added to the vbuf_tree.

Definition at line 1009 of file vion_export.c.

9.28.3.3 vion_dmabuf_unexport()

```
int vion_dmabuf_unexport (
            struct vion_client * vclient,
            vion_buffer_gid_t gid,
            const uuid_t * cred )
```

unexports a dmabuf.

takes as input a couple (GID, credential) produced by a prior export operation. Once unexported the couple (GID, credential) can no longer be imported. It remains alive for all importers that had imported it before the unexport. This function is the core unexport function used by kapi, vbb and uapi to provide the unexport service.

Parameters

vclient	the client which unexports the dmabuf.
gid	the GID of the dmabuf to be unexported.
cred	the credentials of the dmabuf to be unexported, can be NULL.

Returns

- 0: the dmabuf could be successfully exported.
- -ENOENT: the dmabuf referenced by the GID is not (no longer) exported.
- -EACCES: the credentials are incorrect.

Definition at line 1153 of file vion_export.c.

9.28.3.4 vion_dev_ioc_export()

```
int vion_dev_ioc_export (
            struct vion_client * vclient,
            struct vion_export_data * data )
```

exports a dmabuf from user-space.

takes a file descriptor referencing a dmabuf as input, and produces a couple (GID, credential) as output. This function is used by the ioctl function.

Parameters

vclient	the client which exports the dmabuf.
data	the export data structure exposed to the user-space. It contains a file descriptor as intput, a GID and a credential as output.

Returns

- 0: the dmabuf could be successfully exported.
- negative value: indicates the cause of the failure.

Definition at line 1056 of file vion_export.c.

9.28.3.5 vion_dev_ioc_unexport()

```
int vion_dev_ioc_unexport (
            struct vion_client * vclient,
            struct vion_unexport_data * data )
```

unexports a dmabuf from user-space.

takes as input a couple (GID, credential) produced by a prior export operation. This function is used by the ioctl function.

Parameters

vclient	the client which unexports the dmabuf.
data	the unexport data structure exposed to the user-space. It contains a GID and a credential as input.

Returns

- 0: the dmabuf could be successfully unexported.
- negative value: indicates the cause of the failure.

Definition at line 619 of file vion_export.c.

9.28.3.6 vion_vbuf_req_release()

```
int vion_vbuf_req_release (
            struct vion_be * vbe,
            struct vion_buffer * vbuf )
```

requests the exporter to release a dmabuf.

a release request was received by the vrpc layer.

Parameters

vbe	the local vrpc link endpoint the request was received through.
vbuf	a pointer to the vion_buffer representing the dmabuf to be released on the exporter's side.

Returns

- 0: the dmabuf was successfully released.
- EACCES: the dmabuf could not be denied the access.

Definition at line 1253 of file vion_export.c.

9.28.3.7 vion_vbuf_req_import()

```
struct vion_buffer* vion_vbuf_req_import (
            struct vion_be * vbe,
            vion_buffer_gid_t gid,
            const uuid_t * cred,
            u32 * sg_size )
```

requests the exporter to import a dmabuf.

an import request was received by the vrpc layer.

Parameters

vbe	the local vrpc link endpoint the request was received through.
gid	the Global Identifier of the dmabuf to be imported.
cred	a pointer to a uuid_t data structure containig the credentials of the dmabuf, can be NULL is the importer client is trusted.
sg_size	a pointer to a variable containing the size of the dmabuf sg_table, can be NULL if the layout is not needed.

Returns

an ERR_PTR pointer:

- a valid pointer to the vion_buffer representing the imported dmabuf on the exporter's side, if the import can be carried on with vion_vbuf_rsp_import().
- an errno value indicating the cause of the failure otherwise.
  - -ENOENT: the dmabuf does not (longer) exist.
  - -EACCES: the credential are incorrect.

Definition at line 1347 of file vion_export.c.

9.28.3.8 vion_vbuf_rsp_import()

```
int vion_vbuf_rsp_import (
            struct vion_be * vbe,
            struct vion_buffer * vbuf,
            nku32_f gflags,
            struct vion_rsp_import_mult_descr * descr )
```

completes or cancel a dmabuf import.

an import request was received by the vrpc layer.

Parameters

vbe	the local vrpc link endpoint the request was received through.
vbuf	a pointer to the vion_buffer being imported.
gflags	the 'memory-granting-flags' value to be used.
descr	a pointer to a descr data structure to be filled (layout; size), if the import is carried-on, or NULL if the import is to be cancelled.

Returns

- 0: the dmabuf could be successfully imported.
- EACCES: the access could not be granted.

Definition at line 1385 of file vion_export.c.

9.29 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_import.h File Reference

```
#include <linux/version.h>
#include <linux/dma-buf.h>
#include <linux/uuid.h>
#include <linux/ion.h>
#include "vrpc.h"
#include "vion_dev.h"
```

Functions

- int vion_dev_ioc_import (struct vion_client ∗vclient, struct vion_import_data ∗data)
  imports a dmabuf from user-space.
- int vion_dmabuf_import (struct vion_client ∗vclient, unsigned int n, vion_buffer_gid_t gids[ ], const uuid_t ∗creds[ ], struct dma_buf ∗dmabufs[ ])
  imports a set dmabufs.

9.29.1 Detailed Description

vion_import.h - virtual ION driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.lizzi@harman.com)

9.29.2 Function Documentation

9.29.2.1 vion_dev_ioc_import()

```
int vion_dev_ioc_import (
            struct vion_client * vclient,
            struct vion_import_data * data )
```

imports a dmabuf from user-space.

takes as input a couple (GID, credential) produced by a prior export operations, and produces as output a file descriptor referencing the dmabuf build locally. This function is used by the ioctl function.

Parameters

vclient	the client which imports the dmabufs.
data	the import data structure exposed to the user-space. It contains a GID, a credential as input, a file descriptor as output.

Returns

- 0: the dmabuf could be successfully imported.
- negative value: indicates the cause of the failure.

Definition at line 2151 of file vion_import.c.

9.29.2.2 vion_dmabuf_import()

```
int vion_dmabuf_import (
            struct vion_client * vclient,
            unsigned int n,
            vion_buffer_gid_t gids[],
            const uuid_t * creds[],
            struct dma_buf * dmabufs[] )
```

imports a set dmabufs.

takes as input an array of couples (GID, credential) produced by prior export operations, produces as output, an array of pointers on dmabufs in the ERR_PTR format. Valid dmabuf point to dmabuf objects which have been rebuild locally. Invalid dmabufs indicate the cause of the failure. This function is the core import function used by kapi, vbb and uapi to provide the import service.

Parameters

vclient	the client which imports the dmabufs.
n	the number of dmabufs to be imported.
gids	an array of n GIDs.
creds	an array of n credentials, can be NULL.
dmabufs	an array of n pointers to dmabufs,

Returns

- 0: all the dmabuf could be successfully imported.
- -ETIME:
- -EPIPE:

Definition at line 2031 of file vion_import.c.

9.30 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vbb.h File Reference

```
#include <linux/dma-buf.h>
#include <linux/fs.h>
```

Functions

- struct __vbb_context * vbb_context_create (const char *name)

  creates a UBM dedicated kernel-space client.
- int vbb_context_release (struct __vbb_context *ctx)

  release a UBM dedicated kernel-space client.
- int vbb_destroy_buf_id (struct __vbb_context *ctx, unsigned int buf_id)

  unexports a dmabuf.
- unsigned int vbb_create_buf_id (struct __vbb_context *ctx, struct dma_buf *dmabuf)

  exports a dmabuf.

9.30.1 Detailed Description

vion_vbb.h - virtual ION driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.lizzi@harman.com)

9.30.2 Function Documentation

9.30.2.1 vbb_context_create()

```
struct __vbb_context* vbb_context_create (
            const char * name )
```

creates a UBM dedicated kernel-space client.

as any regular kernel-space clients, it has trusted and does not use the strict_import semantic.

Parameters

name	the client name.

Returns

the created client address in the ERR_PTR format.

Definition at line 39 of file vdmaheap_vbb.c.

9.30.2.2 vbb_context_release()

```
int vbb_context_release (
            struct __vbb_context * ctx )
```

release a UBM dedicated kernel-space client.

Parameters

ctx	the address of the client to be released.

Returns

- 0: the client was successfully released.
- negative value: indicates the cause of the failure.

Definition at line 58 of file vdmaheap_vbb.c.

9.30.2.3 vbb_destroy_buf_id()

```
int vbb_destroy_buf_id (
            struct __vbb_context * ctx,
            unsigned int buf_id )
```

unexports a dmabuf.

unlike its corresponding regular kernel-API vion_kapi_unexport() function, it does not take credentials as input parameter.

Parameters

ctx	the address of the client the unexport is issued through.
buf_id	the GID of the dmabuf to be unexported.

Returns

- 0: the dmabuf was successfully unexported.
- negative value: indicates the cause of the failure.

unlike its corresponding regular kernel-API vdmaheap_kapi_unexport() function, it does not take credentials as input parameter.

Parameters

ctx	the address of the client the unexport is issued through.
buf↵_id	the GID of the dmabuf to be unexported.

Returns

- 0: the dmabuf was successfully unexported.
- negative value: indicates the cause of the failure.

Definition at line 88 of file vdmaheap_vbb.c.

9.30.2.4 vbb_create_buf_id()

```
unsigned int vbb_create_buf_id (
            struct __vbb_context * ctx,
            struct dma_buf * dmabuf )
```

exports a dmabuf.

unlike its corresponding regular kernel-API vion_kapi_export(), function, it does not generate credentials for the exported dmabuf.

Parameters

ctx	the address of the client the export is issued through.
dmabuf	the address of the dmabuf to be exported.

Returns

- 0: the dmabuf failed to be exported,
- positive value: the GID assigned to the exported dmabuf.

unlike its corresponding regular kernel-API vdmaheap_kapi_export(), function, it does not generate credentials for the exported dmabuf.

Parameters

ctx	the address of the client the export is issued through.
dmabuf	the address of the dmabuf to be exported.

Returns

- 0: the dmabuf failed to be exported,
- positive value: the GID assigned to the exported dmabuf.

Definition at line 70 of file vdmaheap_vbb.c.

9.31 vdrivers-ref-man-external/linux-kernel-vdrivers/drivers/vlx/vion_vrpc.h File Reference

```
#include <linux/radix-tree.h>
#include <linux/list.h>
#include <linux/mutex.h>
#include <linux/workqueue.h>
#include <linux/uuid.h>
#include <nk/nkern.h>
#include <vlx/pv-version-interface.h>
#include "vrpc.h"
#include "vion_dev.h"
```

Data Structures

- struct vion_rsp_import_mult_descr
- struct vion_vrpc_stats
- struct vion_be

    vion back-end data structure. More...
- struct vion_fe

Functions

- int vion_fe_req_release (struct vion_fe ∗vfe, vion_buffer_gid_t gid, struct vion_import ∗vimp)

    requests the vrpc layer to release a dmabuf.
- int vion_fe_req_import (struct vion_fe ∗vfe, nku32_f gflags, unsigned int ∗nbufs, struct vion_import ∗vimps[ ], const uuid_t ∗creds[ ], int ress[ ], unsigned int flags[ ], vrpc_size_t ∗size)

    requests the vrpc layer to import a set of dmabufs.
- int vion_fe_req_release_flush (struct vion_fe ∗vfe)

    requests the vrpc layer to flush all the pending release requests.

9.31.1 Detailed Description

vion_vrpc.h - virtual ION driver

Author(s): Thierry Bianco (thierry.bianco@harman.com) Christophe Lizzi (christophe.lizzi@harman.com)

9.31.2 Data Structure Documentation

9.31.2.1 struct vion_rsp_import_mult_descr

Definition at line 44 of file vion_vrpc.h.

9.31.2.2 struct vion_vrpc_stats

Definition at line 60 of file vion_vrpc.h.

9.31.2.3 struct vion_be

vion back-end data structure.

represents the exporter's side vion link endpoint. receives requests messages from its peer front-end.

Definition at line 72 of file vion_vrpc.h.

Data Fields

struct vion_device ∗	dev	points to the vion_device this vion_be belongs to
struct mutex	lock	protects this data structure against concurrent accesses
struct list_head	list_node	node for the vion_device list of back-ends
struct vrpc_t ∗	vrpc	points to the vrpc back-end underneath
void ∗	vrpc_data	pointer to the communication memory (i.e. pmem)
vrpc_size_t	vrpc_maxsize	pmem size in bytes
int	vrpc_open	indicates if the vrpc link is open
void ∗	safe_data	points to a safe memory area where import request is copied
vrpc_size_t	safe_size	size of the safe memory area
unsigned int	vrpc_peer_id	vm identifier of the peer vion_fe entity
struct radix_tree_root	export_tree	references by their GIDs all the buffers exported by this vion_be
struct work_struct	peer_off_work	work scheduled when the vlink goes down
struct vion_vrpc_stats	stats	statistics associated with this vion_be
struct pv_driver_header_v1_t	pv_ver	version of this vion_be

9.31.2.4 struct vion_fe

Definition at line 105 of file vion_vrpc.h.

Data Fields

struct vion_device ∗	dev	points to the vion_device this vion_be belongs to
struct mutex	lock	protects this data structure and the pmem against concurrent accesses
struct list_head	list_node	node for the vion_device list of front-ends
struct vrpc_t ∗	vrpc	points to the vrpc front-end underneath
void ∗	vrpc_data	pointer to the communication memory (i.e. pmem)

Data Fields

vrpc_size_t	vrpc_maxsize	pmem size in bytes
unsigned int	vrpc_peer_id	vm identifier of the peer vion_be entity
int	vrpc_open	
int	link_open	
bool	pending_release	
struct radix_tree_root	import_tree	
struct work_struct	peer_off_work	work scheduled when the vlink goes down
struct delayed_work	flush_release_dwork	work scheduled when a timeout expires after a release without import
struct vion_vrpc_stats	stats	statistics associated with this vion_fe
struct pv_driver_header_v1_t	pv_ver	version of this vion_fee

9.31.3 Function Documentation

9.31.3.1 vion_fe_req_release()

```
int vion_fe_req_release (
            struct vion_fe * vfe,
            vion_buffer_gid_t gid,
            struct vion_import * vimp )
```

requests the vrpc layer to release a dmabuf.

dmabuf release can be achieved in two ways:

- immediate release: a release request is immediately transmitted to the peer entity,
- deferred release: the release succeeds immediately and locally, the release request will transmitted to the peer entity later on. When this method is used, all the pending release requests must be transmitted to the peer entity before the next import request, in order to preserve import/release causality. This is done through vion_fe_req_release_flush(). The way release are performed is controlled through the internal compilation flag: VION_DEFERRED_RELEASE_ENABLED.

Parameters

vfe	the local vrpc link endpoint, the request is to be sent through.
gid	the buffer Global Identifier.
vimp	the data structure which represents the dmabuf on the importer's side.

Returns

- 0: if the request was successfully received, processed and answered by the peer entity.
- -ETIME: the request was not answered by the peer entity.

Definition at line 1026 of file vion_vrpc.c.

9.31.3.2 vion_fe_req_import()

```
int vion_fe_req_import (
            struct vion_fe * vfe,
            nku32_f gflags,
            unsigned int * nbufs,
            struct vion_import * vimps[ ],
            const uuid_t * creds[ ],
            int ress[ ],
            unsigned int flags[ ],
            vrpc_size_t * size )
```

requests the vrpc layer to import a set of dmabufs.

the vrpc layer sends an import request to the peer entity, containing the GIDs and credentials for each of the imported dmabuf.

Parameters

vfe	the local vrpc link endpoint, the request is to be sent through.
gflags	the 'memory-granting-flags' value to be used.
nbufs	the number of dmabuf to be imported.
vimps	an array of nbufs pointers to vion_import data structures to be filled after the import (e.g. the dmabuf layout, size, ...).
creds	an array of nbufs pointers to uuid_t data structures containing the dmabuf credentials, can be NULL if the importer client is trusted.
ress	an array of nbufs integers indicating the result of the dmabuf import in the errno format. A zero value indicates that the dmabuf was successfully imported, a negative value indicates the cause of the failure.
flags	an array of nbufs flags containing indications on how the import is to be performed. The only available flag indicates whether the dmabuf memory layout is needed.
size	the size of the import response message. This is only aimed at maintaining statistics.

Returns

- 0: if the request was successfully received, processed and answered by the peer entity.
- -EPIPE: the link with the peer entity is broken. When this happens, all imported dmabuf already imported must be closed.
- -ETIME: the request was not answered by the peer entity.

Definition at line 1123 of file vion_vrpc.c.

9.31.3.3 vion_fe_req_release_flush()

```
int vion_fe_req_release_flush (
            struct vion_fe * vfe )
```

requests the vrpc layer to flush all the pending release requests.

a sequence of release requests is sent to the peer entity (one for each pending request.

Parameters

vfe	the local vrpc link endpoint, the request is to be sent through.

Returns

- 0: all the pending release requests were successfully sent, processed and answered by the peer entity.
- -EPIPE: one pending release request at least could not be successfully sent due to fact that the vrpc link is broken. When this happens, the link is reported as broken, all the dmabuf already imported on this link are invalidated, and must be closed by their owners.

Definition at line 1279 of file vion_vrpc.c.

Index