HARMAN User's Guide

Device Virtualization for Connected Vehicles

Inter-VM Communication Mechanisms

Software Version 12.1

Doc. Rev. 1.2 / 556

Reference: DV-0024

Date: 11/10/2022

Table of Contents

1 Introduction

1.1 This Document

This document briefly describes the inter-VM communication mechanisms delivered with the HARMAN Device Virtualization for Connected Vehicles product. It compares these mechanisms to help the reader understand which mechanism is relevant for his goal. Detailed description of each communication mechanism is available in the "Virtual Device Driver Reference Manual".

1.2 Related Documentation

The Device Virtualization for Connected Vehicles product includes the following documentation:

- Architecture:
  - Product Description
  - System Architecture and Overview
  - Hypervisor Description
- Configuration guides:
  - Hypervisor First Steps
  - Configuring Guest OS and Applications
- User guides:
  - Inter-VM Communication
  - How to Move a Pass-Through Device from System Domain to User Domain
  - Inter-VM Communication for Linux User Space Applications
  - Virtual Driver Writer's Guide
  - Virtual ION Memory Allocator
  - Console Multiplexer
  - Debugging with GDB
  - Hypervisor Awareness in Debugging Environment
  - Suspend to RAM technical report
  - Hypervisor Trace Visualization
  - CPU handover
- Reference manuals:
  - Device Virtualization Reference Manual (Public and/or BSP editions)
  - Virtual Device Driver Reference Manual
- Release Notes

1.3 Support

If you have a question or require further assistance, contact HARMAN's customer support at HCS-DL-RB-FR-Virtualization-Support@harman.com.

2 Hypervisor Services

2.1 NKDDI Basic Services

The HARMAN Hypervisor provides services to support inter-VM communication. These services are available through hypercalls part of the Hypervisor programming interface. These services are made available to Linux Guest OS drivers via the NKDDI library. The full set of API and services is described in the Virtual Device Driver Reference Manual.

The most important basic services are:

- **XIRQ**: eXtended Interrupt ReQuests

  This service enables a Guest OS running in a Virtual Machine to post a software interrupt to another Virtual Machine. This is commonly used to signal a driver within a Virtual Machine (B) that some data or event produced by a Virtual Machine (A) is available.
- **PMEM**: Persistent MEMory

  Persistent Memory may be shared between different Virtual Machines and mapped into their respective Guest OS address spaces. These PMEM chunks are therefore used to exchange data between front-end drivers and back-end drivers without copying data between the Virtual Machines.
- **PDEV**: Persistent DEVice repository

  PDEV services enable to allocate chunks of memory which are private to a Virtual Machine and which can be used to share information with the Hypervisor. These chunks of memory are usually used to store device meta-information by Guest OSes.

2.2 NKDDI vLink

A vLink is a point-to-point communication channel between 2 end-points. XIRQ, PMEM and PDEV resources are usually associated to a vLink. In a typical usage, one end-point is associated to one Virtual Machine while the second end-point is associated to a different Virtual Machine. This allows to easily set up a communication channel between two Virtual Machines.

Each end-point gets its own local PDEV memory chunk to store the local attributes of the communication channel end-point from the Guest OS standpoint. A chunk of PMEM may also be associated to a vLink to enable transmission of data from one VM to the other. Finally XIRQ may be used to signal the other side that some data is ready to be processed.

Therefore, vLinks build on the low-level services and are a nice and convenient way to set up inter-VM communication.

Communication between two vLink end-points may be uni-directional, with a "client" side and a "server" side. However, it is possible to set up bi-directional communication between two vLink end-points. Bi-directional communication can also be set up between two Virtual Machines by establishing two uni-directional vLinks, each using two vLink end-points.

In order for the communication channel to be operational, both end-points must be initialized. Therefore, the vLink service relies on a state machine of the two end-points. This enables one side of the channel to be notified of the "death" of the other side. Communication over the channel can be re-established after the "dead" side comes back.

vLinks are heavily used by other communication mechanisms described in this document.

3 Communication for User Applications

3.1 vBpipe

vBpipe is a Linux device relying on vLink services providing bi-directional communication services between Linux user space applications located in different Virtual Machines. The service provided to user space applications is patterned along the semantics of Unix pipes. A vBpipe can be thought as a pair of pipes.

A vBpipe is usually seen as `a /dev/vbpipeX` devices in the Linux file tree. Access to the service is achieved through regular Linux system calls: `open`, `read`, `write`. A vBpipe enables communication between only two Virtual Machines. Death of the peer application or Guest OS is notified through Unix error codes to the surviving peer.

The service is implemented by a Linux para-virtualized driver. Due to the bi-directional nature of the vBpipe, each driver acts as a front-end and a back-end driver at the same time. Each vBpipe is implemented as a pair of uni-directional vLink, hence featuring four vLink end-points.

Data transfer is similar to what occurs on a Unix system: it is copied from user space to a buffer in the **`PMEM`** area during the `write` system call and is copied from that **`PMEM`** buffer to the user space buffer at `read` time.

Configuration is achieved by declaring appropriate "`vbpipe`" nodes in the device tree.

3.2 vPipe

vPipe is rather similar to vBpipe but provides only a single uni-directional communication channel between Linux user space applications running in different Virtual Machines.

3.4 vRPQ: Virtual Remote Procedure Queue

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.

Data is marshalled to support communication between 32 bits and 64 bits environments. Requests can be described with an IDL (Interface Description Language). The IDL descriptions are processed by an IDL generator which generates code for the client and the server side.

The service is designed for efficiency. Context switches from user space to kernel space is minimized by buffering the requests in a user space buffer as much as possible. Inter-VM communication are limited to the bare minimum: an XIRQ is sent from the client (producer) side to the server (consumer) side upon explicit request from the user application.

The core of the service is implemented as a Linux device. There is a front-end device driver for the client side and a back-end device driver for the server side. Each association between a front-end and a back-end is supported by a vLink.

Data transfer is not symmetrical between client and server sides. On the client side, data is marshaled by the application and written in a user space buffer. Then data is copied from the user space buffer to the vLink **`PMEM`** area. The server side maps the **`PMEM`** buffer directly in its user address space and can thus directly access data without additional copy. This permits to maximize throughput between a client and a server, but obviously at the expense of added latency.

Configuration is achieved by declaring appropriate "`vogl`" nodes in the device tree.

The service has been initially designed to speed up the OpenGL implementation between different VMs, but it is generic enough to benefit other applications.

3.5 vUmem: Virtual User Memory

The vUmem service enables a Linux user space application (client) running in a Virtual Machine to map in its address space a memory area"owned" by another Linux user space application (server) in a different Virtual Machine. When the memory mapping is established, the client may read or write directly in the server's memory area.

Multiple clients running in different Virtual Machines may simultaneously access the same memory area. This concurrency can be prevented if needed.

The service is implemented as a Linux device. There is a front-end device driver for the client side and a back-end device driver for the server side. Each association between a front-end and a back-end is supported by a vLink.

Once the mapping is established, data transfer is immediate. No copy is involved.

Configuration is achieved by declaring appropriate "`vomx`" nodes in the device tree.

The service has been initially designed to speed up the OpenMAX implementation between different VMs, but it is generic enough to benefit other applications.

3.6 vION: Virtual ION Memory Allocator

The virtual ION driver extends Linux/Android ION buffer sharing to inter-VM use cases: it enables clients running in separate VMs to share DMABUF objects backed by local ION memory allocators.

In other words, vION enables an ION buffer allocated within a given VM to be shared with applications hosted by a different VM. The same vION programming interface can be used to transparently share ION buffers between processes that either run on the same Guest OS or on different OSes. The vION services are available to user space application as well as to Linux kernel modules or drivers.

vION buffers are identified by global identifiers (GID) generated when the memory allocator explicitly exports the buffer. These GID can then be used by clients to import such buffers. The import operation returns an open file descriptor which can then be used to memory map the buffer in the client's address space. User space vION services are implemented as `ioctl` operations on the `/dev/vion` device.

A vION buffer can be simultaneously accessed from different Virtual Machines.

The vION Linux kernel driver can act as the back-end as well as the front-end driver. It relies on vRPC services.

Once a remote buffer is mapped in the client's address space, data can be directly read from or written to the buffer without any intermediate copy.

Configuration is achieved by declaring appropriate "`vrpc`" nodes in the device tree with "vion" in the info property.

The vION and the vUmem services are rather similar. However, the vION as an extension of the ION memory allocator so DMA master devices can perform DMA transfer from device to system memory. vION enables a DMA master device owned by VM “A” to produce data in a vION buffer, which can be then transferred to VM “B” and finally can be consumed by a DMA master device owned by VM “B”.

4 Communication for Kernel

4.1 vEth

vEth implements a virtual Ethernet device which can then be used to establish Ethernet based communication as any other Linux Ethernet device. vEth implements a point-to-point Ethernet link between two Virtual Machines. Multiple vEth devices can be configured to establish communication with multiple Virtual Machines. Bridging or routing mechanisms may then be used to transfer frames from one vEth device to another one.

Protocols such as IP, UDP, TCP and others can be used on top of a vEth device.

The implementation is provided by a device driver which implements both front-end and back-end side. It relies on a pair of uni-directional vLink, hence featuring four vLink end-points.

From a Linux user space application standpoint, data transfer is similar to what occurs with a physical Ethernet device. From a sender side, data is copied from user buffer to so-called `skbuf` (buffer used by the Linux protocol stack), and then from the `skbuf` to a **`PMEM`** buffer. On the receiver side, is copied from the **`PMEM`** buffer to a `skbuf` and from the `skbuf` to the user buffer.

Configuration is achieved by declaring appropriate "`veth`" nodes in the device tree.

vEth is also available for QNX and ThreadX.

Note: The Virtio-net(5.2) provides a more usual Ethernet device without the point-to-point limitation.

4.2 vRPC: Virtual Remote Procedure Call

vRPC provides a blocking remote procedure call service for Linux kernel modules. There is also a QNX driver implementing vRPC. No user interface is provided. This enables a client in a Virtual Machine to invoke a "procedure" in another Virtual Machine. The client is blocked until it receives the response from the server.

vRPC instances are named. The name is defined at configuration time. This enables a client to select the server it invokes. The vRPC service is rather simple. Only one call at a time may be performed from a client side to a given server. The maximum size of the data transferred for a call is fixed and defined at configuration time.

The implementation is provided by a kernel module which implements both front-end and back-end side. It relies on a bi-directional vLink.

From a data transfer standpoint, there is no copy of data. The user of the service can retrieve the address of the **`PMEM`** buffer associated to the vRPC object and directly read or write such a buffer.

Configuration is achieved by declaring appropriate "`vrpc`" nodes in the device tree.

This service is used by some other Linux virtual drivers delivered with the HARMAN Device Virtualization for Connected Vehicles such as: vRTC, vClock, vPD and vION.

4.3 vMQ: Virtual Message Queue

vMQ is a Linux kernel module which provides asynchronous message queues. Message queues enable to send one or more messages from a "client" to a "server", and to get responses sent back by the server. Message queues are also bi-directional, hence both sides can act simultaneously as "client" and "server".

Messages are possibly composed of two parts: one fixed size "header" and an optional associated buffer. The "header" is usually rather small and is used to pass control information. Larger data blocks are passed using the optional associated buffer. Messages headers are allocated from a dedicated vMQ message pool. Buffers are associated from another buffer pool. Messages are sent either synchronously, in which case the other side is immediately notified, or asynchronously, in which case the other side is not necessarily notified. The receiving side may use the same message to send back a reply.

The implementation of `vMQ` relies on vLink. To achieve bi-directional asynchronous communication, each side allocates a **`PMEM`** area to store a ring of request, the message pool as well as the buffer pool. As a result, once a "client" has allocated a message, it can directly fill it. No copy will be performed to transfer data to the other side. (Actually, depending upon configuration, data copy of messages may occur in some cases).

There is no explicit configuration of vMQ message queues. As this service is used by other modules, the configuration relies on the configuration of client modules. For example, the vBD service uses vMQ. The name of the vBD vLink is dynamically used to set up the configuration of the vMQ service.

An additional vIPC module is provided to ease the management of multiple simultaneous asynchronous "remote procedure calls", such as issuing multiple concurrent read or write requests to a vBD back-end driver from a vBD front-end driver.

4.4 vIPC: Virtual Inter Process Call

vIPC is a Linux kernel module extending the vMQ service to nicely handle multiple simultaneous asynchronous "remote procedure calls" on top of vMQ. As such, it is dependent on vMQ and cannot be used alone. It can be thought as an extension library to VMQ.

4.5 vEvent: Virtual Event

The `vEvent` service enables to forward events from physical input devices managed by a Virtual Machine to other Virtual Machines. Input devices include keyboards, touchpads, mice and more. Events may be broadcasted to all interested VMs or sent to a single VM (depending on a focus) or dropped.

This service is implemented by Linux kernel *modules*. A back-end module runs aside the real device driver. Front-end modules run in Virtual Machines needing to get input from devices. On the front-end side, availability of input data is signaled by an XIRQ triggered by the back-end module. Access from user space application is achieved through devices created by the front-end modules.

The implementation of vEvent relies on vLink. Transfer of data is achieved through a buffer ring managed in the **`PMEM`** area associated to the vLink. As a result, data is copied from the native driver to the ring by the back-end module and read from the **`PMEM`** area on the front-end side.

Configuration is achieved by declaring appropriate "`vevent`" nodes in the device tree.

4.6 vION: Virtual ION Memory Allocator

This has been covered in section 3.6 vION: Virtual ION Memory Allocator. However, as the vION service also provides a kernel API, it is worth mentioning it here.

4.7 vBD: Virtual Block Device

vBD enables a Virtual Machine to get access to a real disk (hard drive, partition, CDROM…) managed by another virtual Machine. As such it is not really an inter-VM communication mechanism for drivers or user space application, although it moves data from one VM to another one. There is a vBD front-end driver used in the VM needing to access a remote disk. This front-end driver provides a regular disk driver interface to the Linux kernel. On the other hand, there is a back-end device driver which receives and processes requests from possibly various front-end drivers. These requests are forwarded to the local native Linux device drivers using the regular Linux kernel interfaces to access such device drivers.

In the optimum configuration, physical memory pages of a client Guest OS, using the front-end vBD driver, can be "granted" to the back-end driver for the duration of the read or write operation. This enables data to be directly transferred from the client Guest OS to disk for write operations or from disk to the client Guest OS for read operations. Other configuration setups will lead to allocation of "bounce buffers" in the **`PMEM`** area of the underlying vLINK, and to data copies between the two Virtual Machines.

The vBD service relies on the vMQ and vIPC services which themselves relies on vLinks.

The configuration is done by defining back-end nodes in the device tree and referring to such nodes from front-end nodes in the device tree.

5 Virtio

In addition the proprietary inter-VM communication services, the HARMAN Hypervisor provides `virtio` standard interfaces for front-end drivers. It supports console and network devices. These are supported as "`mmio`" (memory mapped input / output) devices.

5.1 Virtio-console

The virtio-console provides a regular console device interface to a Guest OS. The real device (back-end) is provided either the Hypervisor itself. The virtio-console service is implemented directly the Hypervisor. It should be noted that the current Linux implementation of the driver assumes that the virtio-console is mostly used for output and is rather weak for handling input.

Output data is written by the Guest OS driver in the `virtio` buffer shared by the Guest OS and the Hypervisor. It is then copied by the Hypervisor to the real console device.

Configuration is achieved by declaring a "`vl,virtio-console`" node in the device tree.

5.2 Virtio-network

The `virtio-network` service of the Hypervisor emulates an Ethernet device. It enables to set up networks between multiple Virtual Machines. The Guest OS must provide a compatible virtio network device driver. One Virtual Machine may participate in several networks, but each virtual Ethernet device can only participate to one network. Unicast and broadcast are supported.

When sending data in a Linux Guest OS, data is copied from skbuf to the virtual device buffers. Then the Guest OS driver notifies the virtio-network service. The data is then directly copied from the sending Guest OS buffers to available buffers in the receiving Guest OS driver.

Configuration of virtio-network devices is achieved by defining appropriate nodes (vl,virtio-net) in the device tree.

6 Comparison

The following table attempts to summarize the main aspects of the various communication mechanisms. Each row describes briefly a communication mechanism. The columns are detailed below:

- **Goal / Usage:**

  Describe the typical usage of the mechanism, and possibly why it has been designed. (K) indicates that the provided API is for other Linux kernel modules while (U) indicates that a user space API is provided. Some modules provide both.
- **End-Points:**

  Point to point (PtP) communication enables two peers, and only two, to talk to each other over an instance of the communication mechanism.

  "Uni-directional" tells a client side can send data to a server side, while "Bi-directional" indicates that data can flow both directions at the same time.
- **Copies:**

  This list the number of times data is copied to be moved from its origin to its destination.
- **Depends on:**

  Tells whether this mechanism depends on another one.
- **Sync / Async:**

  Tells whether the communication is asynchronous or synchronous. In asynchronous communication a sender is not blocked while receiver is asynchronously notified by a call-back. In synchronous communication, the sender is blocked until its peer sends back a response or an acknowledgement.
- **Lat**:

  Tells whether the communication mechanism is "compatible" with real-time requirements. Actually this rather tells whether the communication mechanism may introduce some latency between the time some data is posted and the time it is processed on the receiving side. To achieve real-time properties in Linux, applications should be running with an appropriate scheduling policy, at the bare minimum.

  No: no latency is added by the communication mechanism.

Name	Goal / Usage	End-Points	Copies	Depends	Sync /	Lat	Remarks
vLINK	(K) Build more high-level communication mechanisms	PtP Uni-directional Bi-directional	None	NKDDI	Async	NO	Convenient wrapper of NKDDI mechanisms
vUPIPE	(U) User applications	PtP Uni-directional	Zero-copy	vLINK	Async	NO	Peer death notification
vRPQ	(U) User applications Batch asynchronous procedure requests Initially developed for OpenGL	Multi End Points Uni-directional Bi-Directional	Sender: User→User User→PMEM Receiver: Read from	vLink	Async Sync	NO	Peer death notification
vUmem	(U) User applications Direct access (mapping) to	PtP Uni-directional	None	XMEM Bridge	Async	?	?
vION	(U) (K) User applications and kernel modules	PtP Nui-directional	None	vLINK	Async	YES	Used by vBD and others
vETH	(K) Kernel Exported as network interface to user applications	PtP Bi-directional	User→skbuf skbuf→PMEM PMEM→skbuf skbuf→User	vLINK	Async	YES	Circular buffer of descriptors in PMEM, data in PMEM
vRPC	(K) Kernel Remote Procedure Call	PtP Bi-directional	Kernel→PME M	vLINK	Sync	NO	Used by vRTC, vClock, VPD and vION.
vMQ	(K) Kernel Library to build other communication	PtP Bi-directional	Copies may be required or not	vLINK	Async	?	Used by vFlash, VBD, vUSB, vI2C, vBufq
vEvent	(K) Kernel Used to transfer input from a physical device	PtP Can be multiplexed	Sender: Kernel→PME M Receiver:	vLINK	Async	?	

Name	Goal / Usage	End-Points	Copies	Depends	Sync /	Lat	Remarks
Virtio.net	Supports virtio API for network. Used for Front-end drivers.	No built-in limitation Bi-directional	?	Nothing	Async	NO	