PikeOS Hardware Virtualization

Am Pfaffenstein 14, D-55270 Klein-Winternheim

PikeOS Hardware Virtualization
PikeOS D5.0, Document Version D5.0-170

SYSGO GmbH
Am Pfaffenstein 14
55270 Klein-Winternheim, Germany

Email: office@sysgo.com

http://www.sysgo.com

Contents

List of Figures

List of Tables

List of PikeOS Pool Elements

1 Introduction

1.1 Purpose

This document explains how the hardware virtualization is implemented in PikeOS and how to use it. The PikeOS Virtualization personality will allow execution of a complete operating system (Linux, PikeOS or another) inside a partition without needing to para-virtualize it.

1.2 Terms and Definitions

This document will use the following terms:

Core thread This is a PikeOS thread in which one virtual CPU core of a guest is executing in. PikeOS sees it and schedules it as any other PikeOS thread.

DirectIO Direct IO access. Feature allowing a guest to access to the hardware directly. This is achieved by a direct mapping of the registers to the guest and the redirection of the interrupt to the guest.

DTB Device Tree Blob. Binary representation of device tree containing hardware and device information in a unified way (e.g. memory areas, cpu clock, interrupt numbers). It is generated from a Device Tree Source (DTS) file with the device tree compiler (dtc).

Guest This refers to the software (usually an operation system) running in a virtual machine.

Host This refers to the software creating and managing a virtual machine. In this case it is PikeOS.

Hardware virtualization This is a hardware feature provided by some processors allowing to create virtual processors to improve the performance when a system wants to run multiple operating systems in parallel.

Hypervisor This refers to the part of the PikeOS kernel managing the processor hardware virtualization support and using it to create and execute hardware virtual machines.

IC Interrupt controlling.

IPA Intermediate Physical Address. This refers to the physical address from a guest point of view. The address is seen as a physical address by the guest but the hypervisor is in fact converting IPA to PA using the hypervisor MMU.

Manager This refers to a PikeOS application creating a PikeOS virtual machine, initializing it and handling actual communication between the virtual machine and the rest of the PikeOS system.

MAP Memory mapping.

MMU Memory Management Unit: Used to translate hardware physical addresses (PA) into virtual addresses (VA). This is used in normal operations to present a process a unified address space in which the real location of the data on the hardware doesn't matter. In case of the hardware virtualization a second stage MMU is used to form the translation between a guest VA to its PA address (IPA).

PA Physical Address. This refers to the real address in hardware or the address seen by an OS or an application when the MMU is not used.

P4 Bus A virtual bus used to abstract device access and provide PikeOS services to a guest. See section 4 for details.

PSP Platform Support Package: Binary code adapting the OS kernel to run on a specific hardware.

VA Virtual Address. This refers to the address an application or an OS sees when the MMU is activated.

Virtual IO This refers to an emulated hardware device. From the guest point of view, it accesses a real hardware device but the device is actually emulated by software on the host instead. This is mainly used to run an unmodified software as guest and emulating parts of the hardware.

Virt-manager Generic name to reference to the PikeOS Virtualization Manager component.

Virtual machine This refers to a hardware virtualized instance on which a guest is actually executing. It provides a virtual CPU to the guest as if it was running on a real CPU. The guest is provided with an interrupt controller and a timer.

VMM Virtual Machine Monitor: Instance controlling and configuring a virtual machine.

2 PikeOS Virtualization

The implementation of hardware virtualization support in PikeOS is divided in two parts: The *hypervisor*, implemented as a part of the PikeOS kernel, and the *manager*, which is a regular PikeOS application using the kernel API to create and manage a guest.

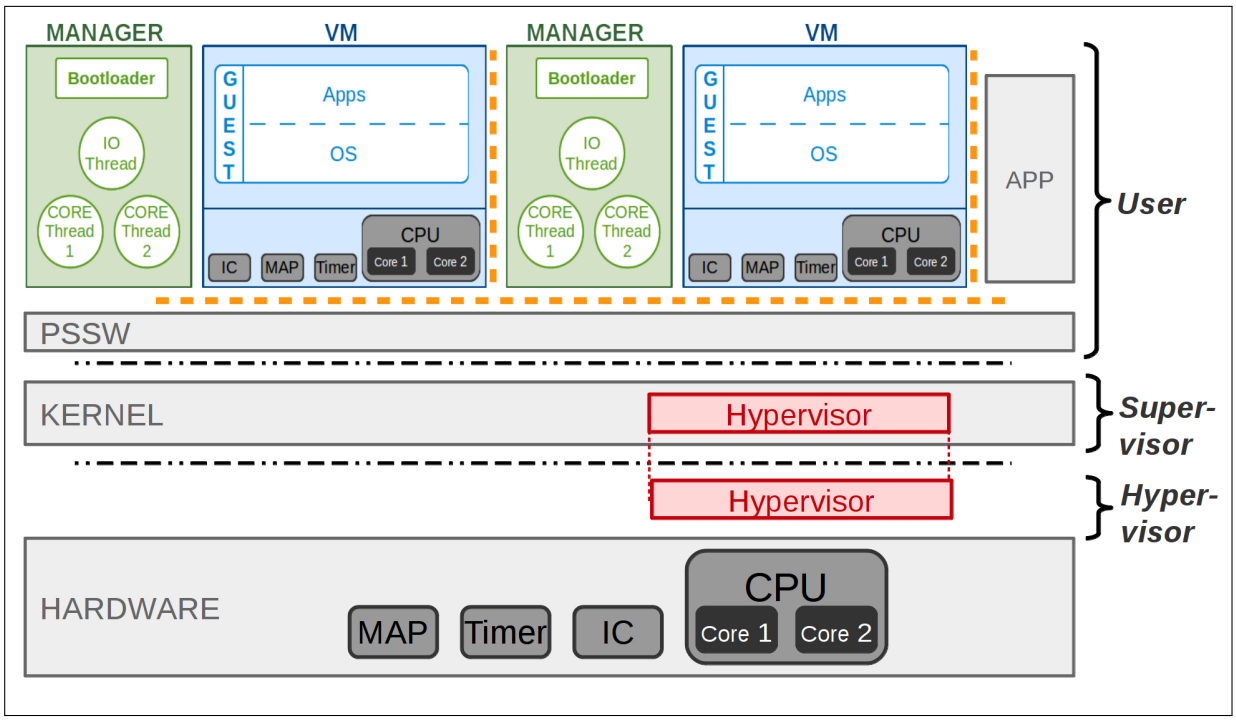

Figure 1: General Design

2.1 The Hypervisor

The hypervisor is the part of the PikeOS Virtualization support implemented directly in the kernel. The main task of the hypervisor is to handle the hardware virtualization support provided by the processor.

In the PikeOS Virtualization implementation, the hypervisor is also handling some other parts of a virtual machine for performance and security reasons. Those are detailed hereafter.

2.1.1 Guest Creation

When a PikeOS application requests to create a new guest, the hypervisor will allocate the required resources and initialize the context to enable the guest's start up later.

2.1.2 Mapping Handling

For a guest to have memory or hardware resource access, a specific mapping table must be created which will be used as a second mapping level. This table will in turn be used by the hypervisor to convert PA to IPA for the virtual machine. This is usually done with support by the MMU.

An address accessed by a guest is converted from VA to IPA using the guest MMU configuration. This IPA is seen as a PA from the guest's point of view and will be converted to a physical address by the hypervisor's MMU configuration. This is usually called a *second level MMU* and is mandatory for the guest to be executed in the same way as it would on the real processor but allowing to actually allocate memory anywhere in the physical address space.

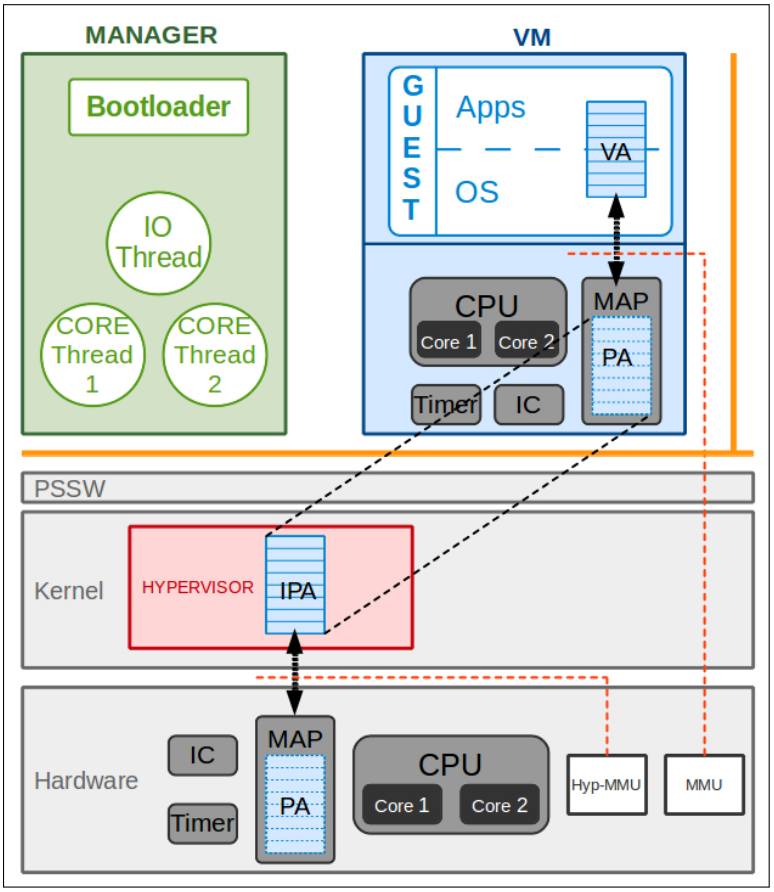

Figure 2: Guest Mapping using Second Level MMU

From a security point of view, this second level MMU also allows detection of an attempt from a guest to access a (IO) memory area which has not been assigned to it. This can be used to virtualize an IO access by generating the access response instead of letting a guest access the desired address directly.

2.1.3 Interrupt Handling

A guest operating system typically requires access to an interrupt controller. As the real hardware interrupt controller is used by PikeOS, we need to have a virtual controller for each guest.

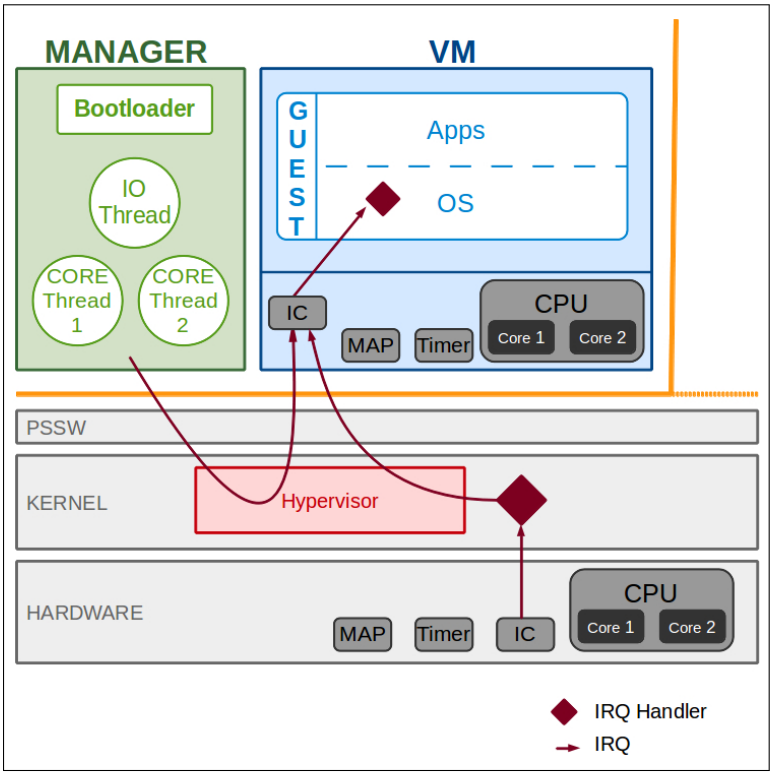

Figure 3: System and Guest Interrupt Flow

The hypervisor provides a virtual interrupt controller to each guest, with help from the processor which usually provides it as part of hardware virtualization support.

The hypervisor will also handle requests from the manager to generate interrupts on the guest (for virtual devices or other communication between the host and the guest).

Finally the hypervisor provides means to forward a hardware interrupt directly to a guest. This is used when a piece of hardware is to be assigned to a guest to reduce the overhead.

2.1.4 Timer Handling

A guest operating system typically requires access to a timer (except some really rare use cases, e.g. bare metal code without needs for a full OS). The hypervisor provides a virtual timer to each guest and is handling the required actions depending on it (for example interrupt generation when the timer expired).

The hypervisor uses hardware virtualization features to provide a virtual timer to the guest. This can be used as a periodic interrupt source or to delay execution (as any timer on real hardware).

It is especially required for sleeping operations. An interrupt is required to wakeup the guest, if it decided to sleep and has given back control to the manager (or generally PikeOS).

2.1.5 Specific Exceptions From a Guest

When a hardware virtual machine is executing, the processor will inform the host using some specific exceptions when the guest has done some forbidden operations or some operations that would usually need handling from the host.

The exceptions generally will be handled by the manager and will only be relayed to the guest if necessary. In some specific cases the interrupts are directly handled inside the hypervior to speed-up the guest's or PikeOS' performance.

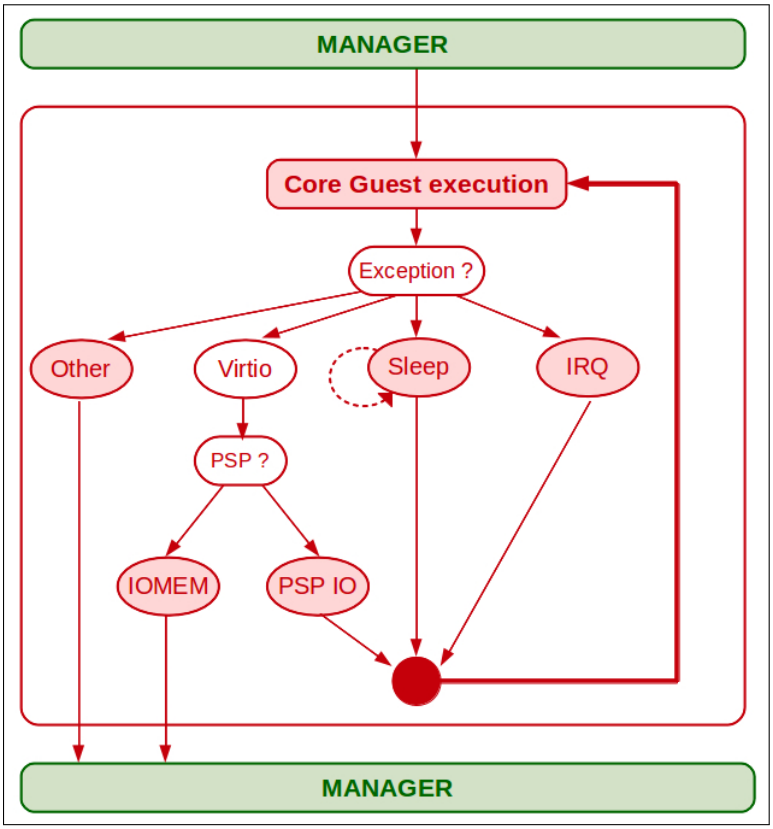

Figure 4: Core Execution Flow in the Hypervisor

The main exceptions handled by the hypervisor are:

- **Processor sleep:** When a guest wants to sleep because it has nothing to compute anymore (usually in the idle task), a specific exception can occur to inform the host. This exception is handled directly by the hypervisor by setting the thread of the given core into sleeping mode from PikeOS point of view, allowing lower priority threads in PikeOS to execute.
- **PSP level virtual IO:** As some hardware devices required to be virtualized are dependent on the board or need some specific hardware access, there is a way to implement virtual IO drivers as part of a PikeOS PSP. This is usually the case for example when a hardware device is used to start other cores of a CPU because this needs special handling in the case of an SMP guest. Those drivers will be called directly by the hypervisor without falling back to the manager.
- **Real hardware interrupt:** Whenever a guest is executing, if a hardware interrupt is occurring, the hypervisor will handle it. This can change the current scheduling and another thread of the system could be executed. In this case the guest will be idled without going back to the manager until it is rescheduled.

In any other case the manager will handle the guest exception.

2.2 The Manager

The manager is a PikeOS user application handling all hardware virtualization related tasks required to create and run a guest inside a virtual machine which are not handled by the hypervisor. The manager is provided as a binary application and will handle the tasks described in the following section.

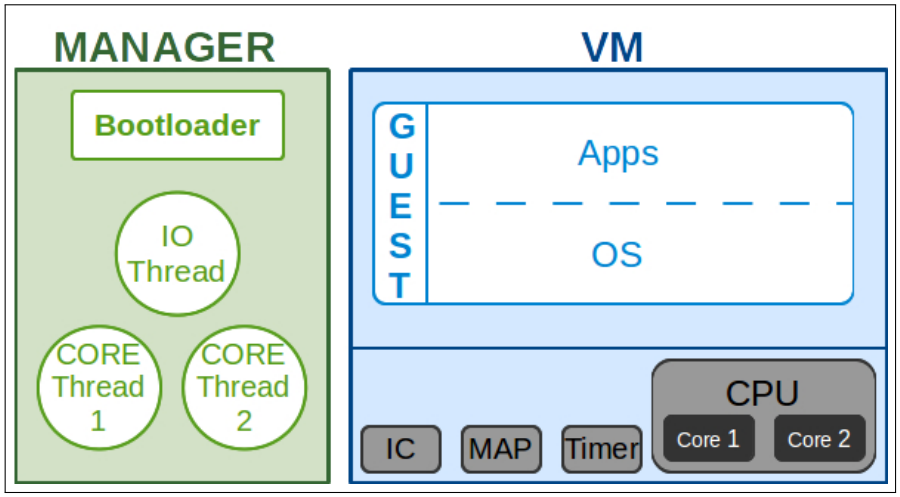

Figure 5: The Manager and its Virtual Machine

2.2.1 Bootloader/BIOS

When an operating system is started on a CPU it actually assumes that a bootloader or at least a BIOS did run before.

This part is simulated by the manager and this includes several tasks:

- **Creating the guest:** This is done by instructing the hypervisor to create a new guest and initialize it.
- **Allocating and mapping memory:** The manager will allocate memory and use the hypervisor to map it into the guest.
- **Loading the guest binary code:** The manager will load the configured binary into the allocated memory and set the guest start address so that it can start executing this binary later.
- **Setting the initial CPU context:** Usually the bootloader is passing some arguments (e.g. command line, memory size, machine ID). The manager will handle this depending on the type of guest that will be executed.
- **Setting IO memory and IRQ mapping:** This is for IO that needs to be mapped and interrupts that need to be redirected directly to the guest.
- **Creation of IO threads:** The manager will create PikeOS threads which will take care of blocking operations inside the manager and communication with other PikeOS services. This feature is mainly used by the P4 Bus.

Afterwards, the manager will create a core thread for each of the guest CPU cores and ask the hypervisor to start the execution of the virtual machine.

2.2.2 Guest Exceptions

Exceptions raised by the guest that are not handled by the hypervisor are forwarded to the manager for further processing.

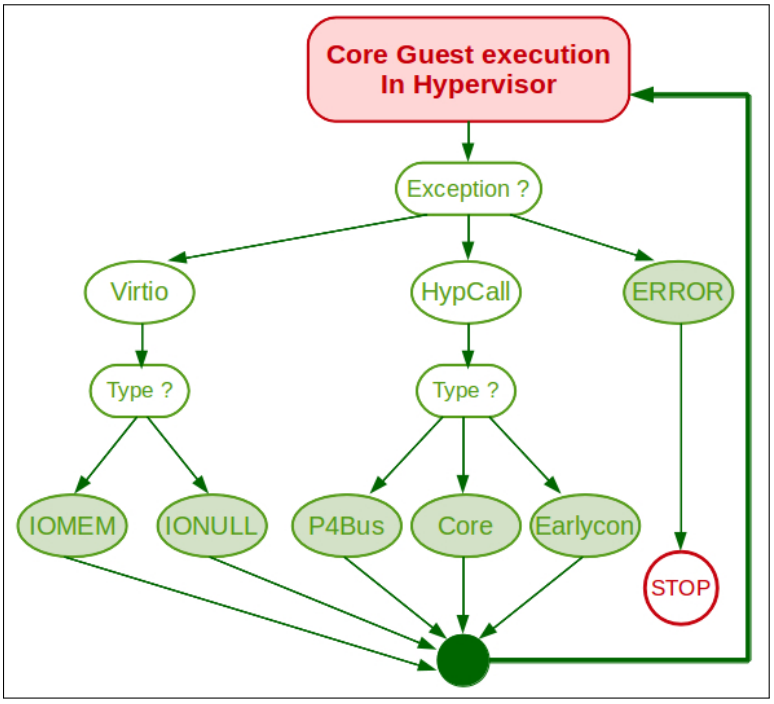

Figure 6: Guest Core Execution Flow in the Manager

The main possible exceptions are:

- **Access to forbidden address:** This can be a forbidden memory address or it can also be used to virtualize an IO access.
- **Attempts to use forbidden processor feature:** Some processor features are not accessible to guests. In most cases those have to be emulated or the guest attempts to do something not authorized.
- **A core has stopped:** The thread corresponding to the core has been deleted by PikeOS or it has been stopped by the hypervisor (due to an error or simply because the guest has asked to stop it).
- **Hypervisor call:** Most hardware virtualization processor implementations provide a specific way to communicate between a guest and a host. The manager is handling these kinds of communication requests (details of possibilities are explained in the next chapter).

The PikeOS manager behavior on those cases can be configured, but the most usual case is to stop the guest if forbidden actions occur.

2.2.3 Guest Communication With the Rest of The System

The communication between the guest and the rest of the system is provided by a specific protocol defined in the generic header **“vmm-def.h”** and implemented in the **“VMM”** layer.

This is the lower communication layer and defines basic VMM operations and VMM devices. The VMM drivers communicate with the host by using Hypervisor Call (HVC) with the specific HVC number “VMM_HVC_NUMBER” (defined to 10) and 4 parameters. These calls are catched and handled directly by the host (**synchronous operations**). By convention, the first parameter is used to indicates the VMM device targeted by the call. The 3 other parameters are used depending of the VMM device.

- VMM_DEV_INFO: Device retrieving system information (memory, CPU available...).
- VMM_DEV_EARLYCON: Device used as an early console by the guest.

- VMM_DEV_BYPASS: Device providing access to an IO area through the Manager.
- VMM_DEV_P4BUS: P4 Bus device used for P4 Bus commands.
- VMM_DEV_VMAPI: Device providing access to vmapi services (reboot the partition, stop the partition,...).

2.3 Guest Scheduling

From PikeOS point of view a guest is a regular PikeOS task with threads which represents the manager application. The threads inside the manager are used to handle communication with the rest of the PikeOS system and there is one thread for each of the guest CPU cores. The guest execution is confined to those core threads. As a consequence the guest execution is subject to the scheduling policy of those threads (priority and possible time scheduling window).

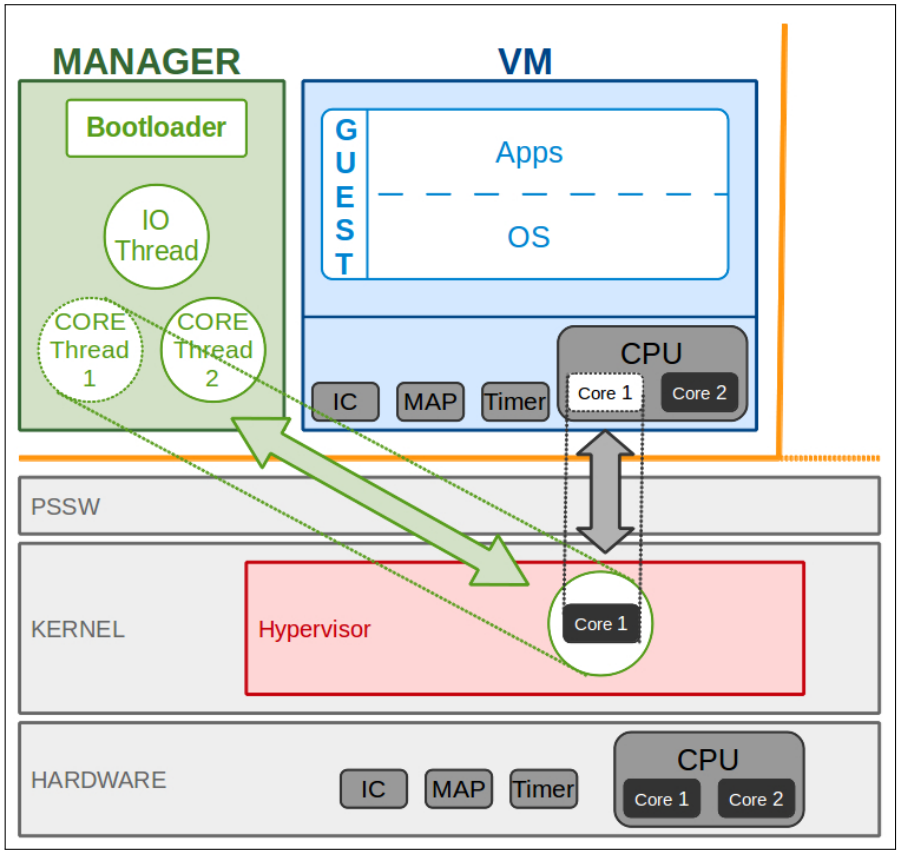

Figure 7: Execution of a Guest Core

When a hardware interrupt occurs while a guest is executing, its execution is stopped, and PikeOS handles the hardware interrupt in the same way it would handle it if any other thread on the system was executing.

2.4 Guest SMP Support

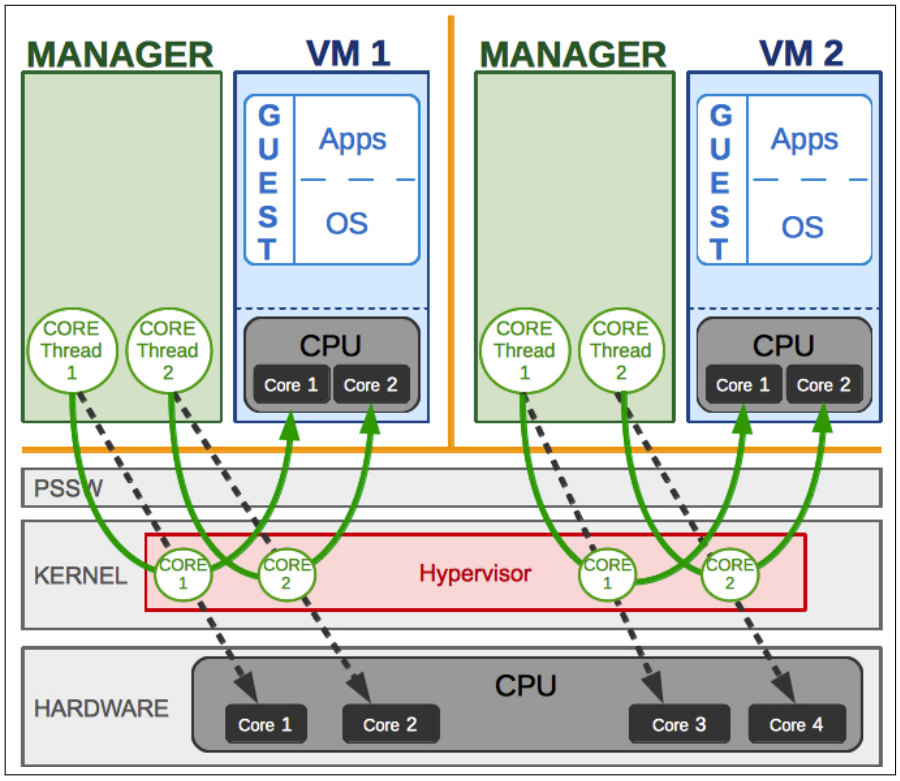

Figure 8: Multi-Core Guests

A guest can use several cores. Each guest core is bound to one host core. For one given guest it is only possible to assign one guest core to one physical core. As a consequence a guest cannot use more cores than the number of cores available in the host.

The manager will start a guest execution thread for each guest core by selecting the next core accessible (depending on the *cpumask* attribute of the task). The available cores are determined by the partition's configuration attribute "CpuMask". This is set in the VMIT.

The following scenario is possible to configure and to be represented on the scheme:

- Guest 1: dual core, guest core 0 running on host core 0, guest core 1 running on host core 1 (partition CpuMask would be 0x3)
- Guest 2: dual core, guest core 0 running on host core 2, guest core 1 running on host core 3 (partition CpuMask would be 0xC).

If the system had only two cores, we could have modified the configuration to execute guest 2 core 0 on host core 0 and guest 2 core 1 on host core 1 as well.

Note: It is not possible to run several cores of the same guest on the same host core.

2.5 Guest DirectIO Support

The DirectIO support for a guest is achieved by two main functions:

- **Registers access:** The physical address of the registers is mapped at the same address in the IPA of the guest. From the guest point of view, the access to these registers is the same as if it was a native operating system running on the target. The mapping between the physical address (PA) and the intermediate physical address (IPA) is managed by the hypervisor MMU. The memory required for the page tables is allocated in the hypervisor memory which needs to be increased if a lot of I/O areas are configured to be accessible directly by the guest (see section 3.5.2). Some guest drivers are DMA compatible which means that they are using some RAM space as IO. In order to avoid trouble with this, in some cases it is preferred to map the guest RAM to its real physical address (see the section 3.5.3).
- **Interrupts forwarding:** As explained in the chapter 2.1.3, the ARM virtualization extension provides a Virtual Interrupt controller which is seen as the real interrupt controller by the guest. In DirectIO, the interrupt attached to the device is forwarded directly to the guest. That means, when an interrupt is raised by the hardware, it is directly seen by the guest through the virtual interrupt controller. The interrupt acknowledgment done by the guest is also directly propagated to the real interrupt controller. The virtual interrupt controller must be mapped at its real physical address for the guest point of view (in the IPA), see the parameter “Use GIC real physical address” in the section 3.5.5.

Note: As the guest is accessing directly the hardware, some registers **must not be configured as DirectIO:**

- **The real interrupt controller**: Access is **forbidden** as the interrupts are managed through the Virtual Interrupt controller. The real interrupt controller is reserved for PikeOS.
- **Reset controller**: Depending of the final use case, accesses to these registers must be done **carefully**.
- **Clocks**: Some clocks modifications can break the PikeOS behavior, accesses to these registers must be done **carefully**.
- **Power Management**: Depending on the final use case, accesses to these registers must be done **carefully**.

Note: The registers which have been configured with DirectIO for a guest cannot be accessible from anyone else on the target. That means that accesses to a hardware cannot be done by **Several guests** OR **PikeOS and a guest**.

2.6 64bit Support and 32bit Guests

Starting with ARMv8, PikeOS also supports Hardware virtualization on 64bit platforms. Together with this, and as long as it is a feature supported by the hardware, PikeOS supports running 32bit guests on top of a 64bit PikeOS.

This is particularly true on ARMv8 where you can run 32bit ARMv7 guests.

All guest drivers and communication protocols are compatible with both 32 and 64bit, so P4Bus and VMM Bus from previous PikeOS are still binary compatible.

3 PikeOS Virtualization Configuration

This chapter describes the different elements available in the PikeOS pool to be used to configure a PikeOS Virtualization guest.

Guests can be created by adding the Virtualization Process to an existing partition or by using the pre-defined groups containing a partition and a process preconfigured for a specific use case.

3.1 System and Configuration Limits

3.1.1 Number and Position of Guests

PikeOS hardware virtualization supports up to *255* guests with only one guest per PikeOS partition. It is not supported to have more than one Guest per PikeOS partition.

3.1.2 Size of Memory (PikeOS 32bit)

All the guest memory is mapped in the manager. As a consequence the guest memory cannot exceed the maximum memory size that can be mapped in the manager. This is theoretically 2GB but as the manager also needs memory, the real size you can allocate to a guest on a 32bit system might be a little bit under this.

More memory can be assigned to a guest but no communication with PikeOS will be possible from this memory. This can be used for Linux guests by adding some extra memory to your guest configured not to be mapped in the manager and used by Linux as high memory.

3.2 Hardware Virtualization Predefined Groups

3.2.1 HWVIRT Linux with DTB

Element Type	Group
Path in pool	virtualization/hwvirt.linux.dtb.dom
Name	HWVIRT linux with DTB

Pool Element 1: HWVIRT linux with DTB

This group contains:

- One virtualization partition
- One virtualization process configured for a hardware virtualized Linux guest using DTB.
- One p4bus device configured as a vmtty (console)

3.2.2 HWVIRT PikeOS

This group contains:

Element Type	Group
Path in pool	virtualization/hwvirt.pikeos.dom
Name	HWVIRT PikeOS

Pool Element 2: HWVIRT PikeOS

- One virtualization partition
- One virtualization process configured for a hardware virtualized PikeOS.

Note: The virtualization process included by this group is configured to run a process from the same architecture of the board. It needs a 32bit PikeOS guest on a 32bit board (ARM v7hf) or a 64bit PikeOS guest on a 64bit board (ARM v8hf).

3.2.3 HWVIRT PikeOS 32bit

Element Type	Group
Path in pool	virtualization/hwvirt.pikeos32bit.dom
Name	HWVIRT PikeOS 32bit

Pool Element 3: HWVIRT PikeOS 32bit

This group contains:

- One virtualization partition
- One virtualization process configured for a 32bit hardware virtualized PikeOS.

Note: This group is only available for the ARM v8hf architecture. It allows running a 32bit PikeOS guest on a 64bit board

3.3 HWVIRT Hypervisor KDEV

Element Type	Component
Path in pool	kerneldriver/hwvirt-hypervisor.cmp
Name	HWVIRT Hypervisor KDEV

Pool Element 4: HWVIRT Hypervisor KDEV

This component is included in all BSP supporting Hardware Virtualization. It is required to be able to add a Hardware Virtualized Guest to your project.

3.3.1 Base Configuration

- **Activate Hypervisor:** Allows to activate or not the hypervisor. If the hypervisor is not activated, no guest can be used.

- **Logging parameters:** It is possible to configure the verbosity of the HWVIRT Hypervisor, depending the version of the version of the kernel driver the customer integrates in the BSP.
  - **Critical:** Only boot errors preventing the kernel boot and usage of the hypervisor.
  - **Info:** Some basic information as long as boot errors, this is the default.
  - **Verbose:** Some extensive information as long as runtime events will be printed. This level should not impact the runtime performance.
  - **Debug:** Extensive information during init and runtime. This level is only available with the debug version of the kernel driver and might have impact on runtime performance. The debug version is providing extensive information when debug or devel logging is selected. This might have a lot of impact on the runtime performance and on boot time. To use this version you must recompile the BSP kernel fusion project and select the hwvirt-hypervisor-debug.kdev binary.
  - **Devel:** Very Extensive information during init and runtime. This level is only available with the debug version of the kernel driver and has a big impact on performance.
- **GIC Registers:** This defines where is the GIC controller as this is needed by the virtual interrupt controller part of the hardware virtualization support. You need here to define the address of the different GIC registers of your board.
  - **BASE address:** On ARMV7, addresses are computed automatically as offsets of the BASE address of the GIC:(DIST = BASE + 0x1000), (CPU = BASE + 0x2000), (CTRL = BASE + 0x4000), (VCPU = BASE + 0x6000).
  - **DIST address:** base address of the GIC Distributor registers area.
  - **CPU address:** base address of the GIC CPU Interface registers area.
  - **CTRL address:** base address of the GIC Control registers area.
  - **VCPU address:** base address of the GIC Virtual CPU Interface registers area.

3.4 Virtualization Partition

Element Type	Partition
Path in pool	virtualization/partition/virtualization.partition.cmp
Name	Virtualization Partition

Pool Element 5: Virtualization Partition

This component contains a generic partition compatible with the Virtualization Process and is used by most of the preconfigured DOMs.

3.4.1 Base Configuration

- **Partition Name:** Name of the partition.
- **Partition ID:** Partition identifier (must be unique on the system).
- **CPU Mask:** Partition CPU Mask. For a hardware virtualization guest this will configure the number of cores of your guest.

3.4.2 Health Monitoring Configuration

Some groups are defined as module of the manager to easily configure Health Monitoring events. Each errors group is detailed in section HWVIRT Errors in chapter 14.

- **Default action:** Health Monitoring default action (for unconfigured events).
- **HYPERVISOR GUEST:** Health Monitoring actions for Hypervisor errors.
- **MANAGER CONFIG:** Health Monitoring actions for Manager Configuration errors.
- **MANAGER DIRECT IO:** Health Monitoring actions for Direct IO module errors.
- **MANAGER FDT:** Health Monitoring actions for FDT module errors.
- **MANAGER GUEST:** Health Monitoring actions for Guest errors.
- **MANAGER VMM:** Health Monitoring actions for VMM errors.
- **MANAGER P4BUS:** Health Monitoring actions for P4BUS errors.
- **MANAGER CPU:** Health Monitoring actions for CPU errors.
- **MANAGER GENERIC:** Health Monitoring actions for generic manager errors.

Note: User can manually setup action to do for each individual error in the VMIT (see PSSW reference manual).
The Health Monitoring event type must be P4_HM_TYPE_UINT and the value is error number (see section HWVIRT Errors in chapter 14).
The group action is done first, groups are like filters to easily configure HM actions to do in error case.
If P4_HM_PAC_IGNORE is selected for the group and a manual configuration exists for the error,
the manual action will be done otherwise default action will be performed.
If the error is tagged as fatal and all configurations are set as P4_HM_PAC_IGNORE, the partition will be automatically shutdown.

3.5 Virtualization Process

Element Type	Process
Path in pool	virtualization/process/virtualization.process.cmp
Name	Virtualization Process

Pool Element 6: Virtualization Process

This component is the main process holding the virtualization manager and it holds generic configuration items for a guest.

3.5.1 Base Configuration

- **Name:** Name for your guest (used for partition process name).
- **Type:** Type of your guest, can be Linux, PikeOS or Generic.
- **32 bit guest:** Defines if the guest is a 32bit guest. 32bit guests and 64bit guests are supported on an armv8 host.
- **Log Level:** This defines the verbosity of the manager (mainly on start up but also when a guest generates an exception). “info” level will print almost nothing except on error conditions and “debug” level will tell you almost anything happening.
- **Guest binary:** Path of the guest binary image. The path could be relative to the filesystem, the CUSTOM_POOL configured for the project or the PIKEOS_POOL.
- **Binary load offset:** Offset in the guest RAM at which the binary is loaded.

- **Start offset in Ram:** Offset in the guest RAM at which the guest is started.
- **Manager Priority:** Maximum priority of the manager, the guest cores are executing at this priority - 3.

3.5.2 Host Memory

- **Manager memory size:** Size of the memory used by the manager as a memory pool for the hypervisor. It could need to be increased depending on the features you activated for your guest.
- **Kernel memory size:** This is the partition KMEM. This could need to be increased if you increase guest or manager memory size.
- **Hypervisor memory size:** Size of the pool allocated and used for the page tables of the guest.
- **Hypervisor provider name:** Name of the Hypervisor provider when using a Partition Integration Project.

3.5.3 Guest Memory

- **RAM size:** Size of memory allocated and used as guest RAM. The size must be a multiple of 1 MB or 2 MB depending of the guest.
- **RAM alignment:** Alignment of the guest RAM.
- **Use real physical addresses:** Allows the guest to use physical addresses. By activating this option, the intermediate physical address will be the same as the physical address and second level of MMU is only used to control access rights of the guest. If a guest is configured to directly access devices using DMA (DirectIO component) and does not see the real physical address, it will not be able to use the driver doing DMA properly.
- **RAM guest physical address:** Address of the guest RAM for the guest point of view (available if “Use real physical addresses” is unset).
- **Activate shadow mapping:** Allows you to have a second address from which the memory can be seen by the guest. This can be usefull to have high address memory also seen at a lower address.
- **Guest Shadow address:** address of the second mapping.
- **Automatic allocation:** Allows the PSSW to allocate the RAM.
- **RAM host physical address:** Address of RAM on the host (available if “Automatic allocation” is unset).

3.5.4 Guest virtual Memory

When a PikeOS Hardware Virtualization guest needs to map a resource (for example by using a mmap operation though the P4 Bus), the PikeOS Virtualization Manager uses a free physical memory area to provide the requested resource to the guest.

- **Use real physical address:** Defines if the manager will map the resource at its real physical address (option set to true) or if the PikeOS Virtualization Manager will use a pool of free physical addresses for the guest, for which there is no RAM or registers mapped (option set to false).
- **Virtual memory addresses:** Address of a guest free physical addresses pool (available if “Use real physical addresses” is unset).
- **Virtual memory size:** Size of the guest free physical addresses pool (available if “Use real physical addresses” is unset).

3.5.5 Interrupt Controller

- **Use GIC real physical address:** Allows the guest to use the GIC real physical address. For a DirectIO guest this option should be set to true. If set, the Virtual Interrupt controller will be mapped to the guest at the real interrupt controller address. For the guest, it will access to the real interrupt controller which will be in reality the virtual interrupt controller.
- **GIC Controller Dist address:** GIC Distributor physical address for the emulated GIC controller (available if “Use GIC real physical addresses” is unset).
- **GIC Controller CPU address:** GIC CPU physical address for the emulated GIC controller (available if “Use GIC real physical addresses” is unset).
- **GIC IIDR:** Implementer Identification Register value(default value 0x0200043b is for a GICv2 from ARM). Hardware Virtualization currently only support GICv2 simulation for a guest, so be careful when changing this value to a have the correct product ID.
- **GIC TYPER ITLinesNumber:** number of interrupt line number to simulate, default 0x1f corresponds to 1020 interrupts available for a guest. For more information you can check the "GIC architecture specification" from ARM.

3.5.6 Virtual Watchdog

- **Enable watchdog:** Allows the manager to use the Virtual Watchdog to monitor the running status of the Guest and to forward to the Health Monitoring the exception to raise in case of a dead Guest.
- **Autostart watchdog:** Allows the watchdog to be started when the manager start the Guest process (available if “Enable watchdog” is set). Usefull to check that constraints, if any, on boot time is respected.
- **Startup Timeout:** Timeout value used in the case of a Autostart watchdog (available if “Autostart watchdog” is set). Value is in nanoseconds.
- **Running Timeout:** Timeout value used during the running phase of the Guest (available if “Enable watchdog” is set).

Note: There are three possible use cases, whether for a Linux Guest or a Pikeos Guest:

- **Austostart:** As soon as the Manager start its drivers, including the wmm-watchdog, if the watchdog is enabled and the autostart is enabled, it starts its countdown using the **Startup Timeout** as first timeout and then, when refreshed, use **Running Timeout** as countdown.
- **Start on device open:** Once the Manager is started and as soon as the Guest open a file descriptor on the watchdog device.
- **Start on Command:** The watchdog is commandable through the vmm commands VMM_WDT_START, VMM_WDT_STOP, VMM_WDT_REFRESH, VMM_WDT_GET_TIMEOUT, VMM_WDT_SET_TIMEOUT.

3.5.7 Virtual IO

Here you can configure the Virtual IO logging. The virtual IO tracing can be used to have a record of all accesses from a guest to one or several IO areas in order to analyse a guest driver behaviour. A log will be generated of all read and write accesses to the define Virtual IO areas with the address accessed, the value written or read, the access size and the PC when the access was done.

- **Enable Virtual IO logging:** enable the VirtIO logging.
- **Log output:** The generated logs can be stored on different places:
  - On the PikeOS console.

  - On a file provider. This can be used for example to store the logs on a muxa channel to have them directly on your development host.
  - On a file. This can be used to store the logs on a shared memory for later retrieval using the PikeOS monitor.
  -

- **Storage File Path:** path of the Log output file.
- **Storage File Path:** When logs are stored on a muxa channel or transfered via network, it can be better for the performance impact to bufferize the logs and send them grouped. The buffer depth allows to group log entries before writing them to the storage file. When using a muxa channel to transfer the logs, a value of 7 will fit in one muxa packet.

3.5.8 Debugging

- **Boot Delay:** If this value is not 0, before starting the guest, the manager will sleep for X seconds, X being the parameter value. This can be used to delay a guest start and prevent having several guest log messages on start-up or prevent a guest from starting before one of the (PikeOS) drivers it depends on is started.
- **Debug Print Frequency:** If this value is not 0, for each of the guest cores the current execution address will be printed on the PikeOS console every X milliseconds, X being this parameter's value. This can be used to follow early guest boot, if there is no console available, but should not be activated otherwise.
- **Allow debug call:** Allows handling of a specific HVC/SMC call to print the guest context without stopping its execution. The r0 (or x0 on arm-V8) register must be equal to 0xffffffff before to do the HVC/SMC call.
- **Stop on IO error:** If selected, any attempt to access a forbidden physical address by a guest will stop it directly, print the accessed hardware address and the guest execution address on the PikeOS console. When this option is not activated, accesses to forbidden addresses are ignored for write and return 0 for read.
- **Stop on CP15 error:** If selected, any attempt to access a forbidden coprocessor register by a guest will stop directly the guest. If not selected, an invalid coprocessor read will return 0 to the guest and a write will simply be ignored. Main coprocessor registers not accessible to a guest are those related to performance counters, secure features or hardware virtualization functions. This options only exists for ARM processors.

3.5.9 Linux Boot

This section is specific for Linux guests.

Linux Boot configuration:

- **Linux Command Line:** These parameters allow configuration of the Linux kernel command line used by the guest.
  - Type: Indicates the command line source type. It can be set to "None", "File" or "String".
  - Command Line file: Path of the command line file. Please select first the root of the path (PIKEOS, PIKEOS_POOL, CUSTOM_POOL or FILESYSTEM).
  - String: String used as command line (available if Type is set to "String").
- **Device Tree (DTS/DTB):** Device Tree configuration. It is only available for a Linux guest.
  - DTB Support: Activates the DTB support.
  - Allow the Manager to patch the DTB: Allows the manager to modify the DTB on specific nodes (memory, command line, CPUs). If this option is unset and your Linux guest contains some P4 Bus devices, please define the "CONFIG_VMM_ENUMERATE" symbol in your Linux kernel configuration, see the section 7.
  - Dump DTB: Dump the content of the DTB including modifications before starting the guest.
  - DTB File: Path of the DTB file. Please select first the root of the path (PIKEOS, PIKEOS_POOL, CUSTOM_POOL or FILESYSTEM).
  - Load Offset: Offset of the DTB in the guest memory.

3.5.10 P4Bus

- **Parameters:** P4BUS Operation Threads.
  - **Number of Operation threads:** Number of threads to create for the P4 Bus devices operations.

Note: The number of operation threads should not be too small, otherwise your guest will not be able to use several devices in parallel. You can find out how many threads you may need by counting the number of blocking calls (i.e. an Ethernet device would need two threads, a console would also need three threads) and add some extra threads for non blocking calls (i.e. sampling ports, open or close calls, etc.).

In order to evaluate the number of operation threads required per guest, you can refer to the following table. Please add at least two operation threads more than the result of your evaluation. These are required for OPEN/CLOSE/IOCTL operations.

Guest type	P4 Bus guest driver	Number of operation threads required **per devices**.
Linux	p4bus-vmtty	3
Linux	p4bus-vmnet	2
Linux	p4bus-vmblock	1
Linux	p4bus-vmchar	X, the value is depending of the number of blocking operations done in parrallele on the device.
PikeOS	vmm-console	1 per core configured for the guest (CPUMASK).
PikeOS	vmfp	X, the value is depending of the number of blocking operations done in parrallele on the device.

Table 1: Operation threads number evaluation

3.5.11 IOMMU

- **Parameters:** Enable IOMMU.
  - **Description::** If checked the IOMMU driver component assigned to the Guest shall use the Hardware Virtualization Guest Translation Table to reserve memory for the Guest.

3.5.12 Registers Value on Boot

This allows the integrator to modify the registers values when the guest starts. For each register you can set the following options:

- **Type:** The type of the value. It can be set to “unmodified value” , “Add RAM Physical Address (IPA)”, “Add RAM Real Physical Address (PA)”
  - unmodified value: The value will be handled as a RAW value, without any interpretation.
  - Add RAM Physical Address (IPA): The guest physical address is to be added.
  - Add RAM Real Physical Address (PA): The real physical address of the guest is to be added.
- **Value:** The value of the register.

Note: For a 64bit Linux guest using the DTB, the x0 register is set automatically and contains the DTB address. The others are not used for a basic Linux kernel but can be used for advanced debug investigations.

Note: For a 32bit Linux guest, the following registers are used at boot time:

- **r1 register:**
  - The r1 register is forced to 0 to inform the guest to use the DTB.
- **r2 register:**
  - The r2 register contains the DTB address.
- **r4 register:**
  - The r4 register contains the guest start address.
- **r5 register:**
  - The r5 register contains the guest start address.

3.5.13 ARM Parameters

This section allows activation of some of the ARM core features.

- **Debug:** This option allows the guest the access to the Debug core registers. It could need to be increased depending on the features you activated for your guest. This option is required by PikeOS on armv8 and Linux as hardware virtualization guest.
- **Performance counters:** This option allows the guest the access to the Performance Monitor Unit registers. The Performance Monitor Unit provides performance counters whose the values are updated every cycle or every 64 cycles (depending of the PMU configuration).
- **Trace:** This option allows the guest the access to the Trace core feature.
- **FPU/SIMD:** This option allows the guest the access to the Floating Point Unit or/and to the Single Instruction, Multiple Data unit. This option is required by a hardware virtualization Linux guest and by PikeOS if any application is using the FPU.

3.6 Virtualization File

Element Type	Sub-component
Path in pool	virtualization/misc/virtualization.file.scmp
Name	Virtualization File

Pool Element 7: Virtualization File

This sub-component can be used to load the content of a file at a specific place in the guest memory before starting it. This can be used for Linux Initrd for example.

3.6.1 Base Configuration

- **Target Name:** Name of the file on the PikeOS ROM file system (must be unique).
- **Host File:** Path of the file. Please select first the origin on which the path will be relative (PIKEOS, PIKEOS_POOL, CUSTOM_POOL or FILESYSTEM).
- **Load offset:** Offset in guest RAM at which the file must be loaded.

3.7 p4bus device

Element Type	Sub-component
Path in pool	virtualization/p4bus/p4bus.device.scmp
Name	p4bus generic device

Pool Element 8: p4bus generic device

This sub-component can be used to create P4Bus devices and to be able to set custom guest types. This is used to create new P4Bus guest drivers or when porting the P4Bus to new guest types, for more detailed information on P4Bus devices see the section 4.

3.7.1 Base Configuration

- **Device Name:** Name of the device (must be unique for a guest).
- **Host Type:** Host type of the device. Can be vmcprintf, vmfile, vmqport, vmsport or vmnull.
- **Use a File Provider:** When the Host Type is vmfile, this creates a dependency on a character device that will be used to retrieve the file to be opened on the host. This can be used when you want to connect a p4bus device to an ethernet driver, a MUXA channel or a serial driver for example. The required file access is added automatically.
- **File access mode:** When the Host Type is vmfile, the file access mode should be selected. The default value is Read Write Map. The different accesses available are 0 = No Access, 1 = Read Only, 2 = Write Only, 3 = Read Write, 4 = Read Write Map.
- **PikeOS File:** When the dependency system is not used, this parameter allows giving the complete path to the file to be opened for this p4bus device. This can be used to access some file on the rfs file system or a shared memory. No file access will be added.
- **Guest Type:** Type of the device on the guest side.
- **Guest Name:** Name of the device on the guest side.
- **Enable Network Emulation:** Network emulation can be activated to provide to the guest (or to the host in case of vmfprov) a mac address if it isrequesting one.
- **MAC Address:** MAC address of the device to be emulated. I f you let a 0 mac address, one will be generated using the following formula:
  - 56:49:52:devid:partid[7..0]:partid[14..8],[0|1]

  In this formula:
  - **56:49:52:** corresponds to the letters v,i, and r in hexadecimal.
  - **devid:** device number.
  - **partid[7..0]:** are the lowest bits of the partition ID of the guest.
  - **partid[14..8]:** are highest bits of the partition ID.
  - **[0|1]:** lowest bit of the last field is 0 for the guest address and 1 for the host address (in case of a vmfprov device).

  This is compatible with host types vmfile, vmqport and vmfprov.
- **Ports configuration:** depending on the Host Type of the configured P4 Bus device, i.e. vmsport or vmqport, you can set the ports configuration:

  - **use existing ports:** You need to create the ports manually.
  - **not connected:** Ports will be created automatically using the Device Name.
  - **interconnected:** Ports will be created automatically and connected each other using the Device Name.
  - **connected to an other p4bus device:** Ports will be created and connected automatically between the slave p4bus device and the other one.

- **Slave p4bus device:** By checking this option, this device will inherit parameters from the other p4bus device. Ports and channel will be automatically created. IF you want to connect two devices, one of them needs to activate this option.
- **Max message size:** Configures the maximum size of the message.
- **Max message count:** Configures the depth of the queuing port.
- **Refresh rate:** Configures the refresh rate of the sampling port.
- **Source Port:** When vmqport or vmsport type is chosen, this is used to give the name of the source port.
- **Destination Port:** When vmqport or vmsport type is chosen, this is used to give the name of the destination port.

3.8 Virtualization virtual IO

Element Type	Sub-component
Path in pool	virtualization/misc/virtualization.virtio.scmp
Name	Virtualization virtual IO

Pool Element 9: Virtualization virtual IO

This component shall be used to simulate all access to a predefined IO area using either memory (initialised to 0), a NULL area (all writes are ignored, all reads return 0) or a hardware register area.

Using this component you can map put registers, memory or a null output (read return 0 and writes are ignored) in front of a guest physical address range.

This can be used to stub some registers you don't want to give access to the guest or to log all accesses of a guest driver you want to analyse (using bypass mode and the Virtual IO logging). This can be used also when a non modified binary is to be used as guest executable to let some drivers initialize even if the peripheral is not accessible.

3.8.1 Base Configuration

- **Name:** Name of the virtual IO. This name might be displayed in logs to identify accesses.
- **Type:** Type of the virtual IO:
  - **Null:** this is just ignoring writes and returning 0 to reads on the area.
  - **Memory:** this is allocating some memories so that values written are read back by the guest.
  - **Bypass:** this is accessing some real registers and forwarding accesses from the guest to them.
- **Physical Address:** Guest physical address for the virtual IO. An area does not need have its address or size aligned to a page, this can be used to restrict a guest rights to a subset of a page using a bypass area.
- **Guest Intermediate Physical Address:** In case of a bypass area, the guest IPA can be set to a value different of -1 to have a different address on the guest then the real physical address. This can be used for example to map a different serial line to the registers of the first serial line to have the guest console on a secondary serial when present.

- **Size:** Size of the virtual IO.
- **Read Only:** With the Read Only parameter you can let the guest access in read mode to the area but all write access from the guest will be ignored silently.
- **Log Accesses:** When Virtual IO logging is enabled on the manager, this parameters is used to select which Virtual IO areas will actually log accesses.

3.9 Virtualization memory

Element Type	Sub-component
Path in pool	virtualization/misc/virtualization.memory.scmp
Name	Virtualization Memory

Pool Element 10: Virtualization Memory

This sub-component can be used to add a memory region to a guest. This memory region can be configured to be mapped or not in the Manager memory. If not, it can be set as an IOMEM by specifing a physical address

Note: LPAE Memory: This sub-component can be used to declare an LPAE region that can be used by a guest. For this use case the Virtualization memory must be configured as IOMEM by giving the real physical address of the LPAE memory. For a Linux guest, the LPAE and HIGHMEM support must be activated in the kernel configuration.

3.9.1 Base Configuration

- **Memory name:** Name of the memory.
- **Memory size:** Size of the memory.
- **Memory alignment:** Alignment of the Memory.
- **Use real physical addresses:** Allows the guest to use physical addresses.
- **Memory guest physical address:** Address of the memory for the guest point of view (available if “Use real physical addresses” is unset).
- **Activate shadow mapping:** Allows you to have a second address from which the memory can be seen by the guest. This can be usefull to have high address memory also seen at a lower address.
- **Guest Shadow address:** address of the second mapping.
- **Map the memory in the manager:** This option allows the manager to map the memory region in its own memory.
- **Set this memory as IOMEM:** This option allows configuration of the memory as an IOMEM region instead of a RAM region (available if “Map the memory in the manager” is unset).
- **Automatic allocation:** Allows the PSSW to allocate the memory.
- **Memory host physical address:** Address of Memory on the host (available if “Automatic allocation” is unset).

3.10 HWVIRT custom direct IO

This component shall be used to grant direct access for a hardware virtualized guest to a predefined IO area and/or interrupt.

Element Type	Sub-component
Path in pool	virtualization/hwvirt/hwvirt.directio.custom.scmp
Name	hwvirt custom direct IO

Pool Element 11: hwvirt custom direct IO

3.10.1 Base Configuration

- **Device Name:** Name of the custom direct IO entry.
- **Registers address:** Base address of the registers to grant access to (must be page aligned):
- **Registers size:** Size of the IO area to grant access to (if 0 no registers access will be granted).
- **Read Only:** Select this if you want the area to be read-only for the guest.
- **Cacheable:** Set the IO area as cacheable for the guest.
- **Interrupt number:** Interrupt number to forward directly to the guest (if 0 no interrupt is forwarded).

Note: By default IO registers must not be mapped cacheable to a guest as it could make it possible for a guest to block the system. If IOs are accessed with cache enable on ARM, the system could get blocked when the cache will be flushed. You should only set the **Cacheable** parameter for IO areas which are not registers (like external memories).

3.11 VMIT

For each guest you add to an integration project, a partition will be added with one process to the VMIT.

The following elements are of interest on that partition and could need modifications:

- **Partition MaxPrio:** If you want to change the guest priority on the system you will have to change the processes' maximum priority and the partition maximum priority accordingly.
- **FileAccessList:** Any file you want your guest to access, must be accessible for the partition.
- **Process MaxPrio:** The guest core(s) will be executed at *MaxPrio-3*. You must adapt this value depending on your needs.

4 P4 Bus Communication

The P4 Bus has been introduced with PikeOS Virtualization support to provide a generic way for the communication between a guest and the underlying PikeOS. The P4 Bus is available with PikeOS Hardware Virtualization. The basic concept is to abstract both guest and PikeOS communication systems through a generic bus allowing to connect anything on one side to anything on the other side.

For example you could connect a Linux Ethernet device to a set of queuing ports in PikeOS or to a file provider without impacting Linux's side, if the type of the PikeOS object is changed.

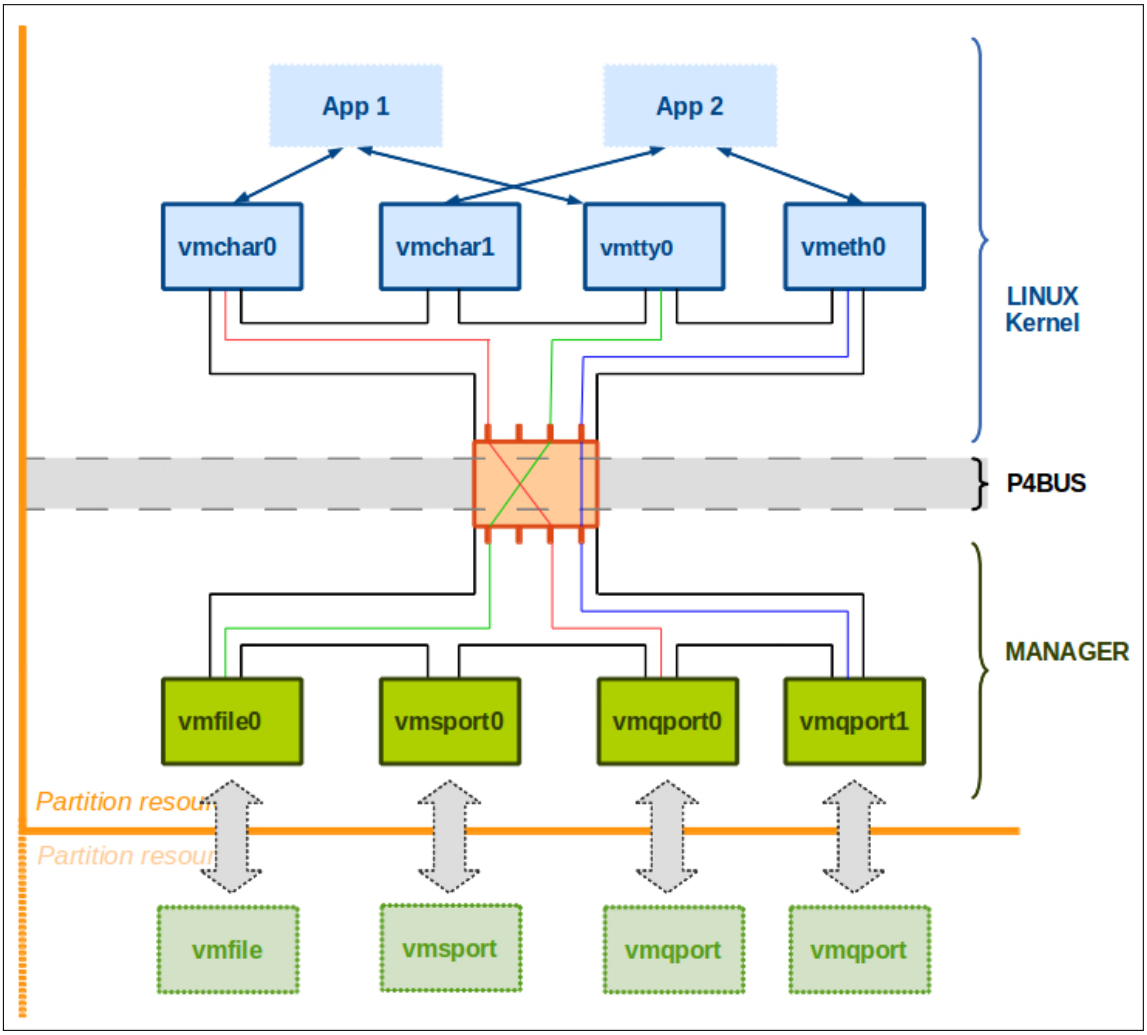

Figure 9: P4 Bus Overview

4.1 P4 Bus Protocol

The P4 Bus is designed to have the guest on one side, requesting operations, and on the other side the host getting operation requests, processing them and providing the results of the operations back to the guest.

This bus has been designed to be as generic as possible to allow implementing the low level part actually doing the communication in several ways without impacting the bus client on both sides.

The P4 Bus host side is implemented directly into the manager. This is mandatory to allow communication with PikeOS system software features (e.g. files, ports) and other applications running in different partitions of the host system.

As a consequence the communication rights of a guest are the ones of the manager from this specific guest (e.g. file access and ports). This is designed to restrict a guest by the rights of the partition of its corresponding manager, this is valid for space (e.g. IO, abilities and resources) and time partitioning.

The P4 Bus in the manager has a pool of operation threads to actually execute the PikeOS operations and to be able to handle several blocking operations in parallel.

Note: Since PikeOS-4.2, the communication layer between the guest and the manager has been completely reworked to be fully 64bit compliant and to improve the performances. It introduces the following basic changes:

- The memory area shared between the guest and the manager (P4 Bus operation and IORING) is statically allocated in the guest and transmitted to the manager. As this area is in the guest memory, it is automatically used as cached.
- IPI are not used anymore to signal the guest when an operation is done. For each P4 Bus device declared for a guest, the manager is checking for a free IRQ and attaches it to the device. This interrupt is used to signal the end of an operations to a P4 Bus device. One interrupt is attached to one p4bus device.

The communication between the Guest and the Host is based on 2 layers, the VMM layer described in the section 2.2.3, and the P4 Bus layer.

The P4 Bus layer is using the basic services provided by the VMM layer. It is defined by the generic header **“p4bus-def.h”** and provides the generic commands, types and operations that can be used by the P4 Bus drivers. The P4 Bus commands are specific VMM operations (**synchronous operations**) with the VMM device ID “VMM_DEV_P4BUS” and a command ID as first parameter. The 2 other parameters are used depending of the command.

- P4BUS_COMMAND_INFO_VERSION: Retrieves the P4 Bus protocol version.
- P4BUS_COMMAND_INFO_DEVICE: Retrieves P4 Bus device information for a given ID.
- P4BUS_COMMAND_CREATE_IORING: Informs the manager to attach and start an operation thread to the given IORING.
- P4BUS_COMMAND_DESTROY_IORING: Informs the manager to detach the operation thread for the given IORING.
- P4BUS_COMMAND_SIGNAL_IORING: Informs the manager that there is an operation in the IORING to execute.
- P4BUS_COMMAND_EXECUTE_OPERATION: Send a request to the manager to execute an operation.
- P4BUS_COMMAND_CANCEL_OPERATION: Informs the manager that the given operation must be canceled.

4.1.1 P4 Bus drivers compatibility

The Linux Hardware Virtualization drivers are available in ELinOS and in PikeOS. User can choose use Virtualization drivers version provided with PikeOS instead of ELinOS. Please refer to the section 12 for more information.

The following table describes the compatibility between PikeOS and ELinOS version using ELinOS P4 Bus drivers. Theses drivers could be found in Linux sources directory in ELinOS directory in folder drivers/virt/pikeos.

PikeOS / **ELinOS**	6.0.X	6.1.X	6.2.X	7.0.X
4.0.X	Compatible	Compatible	Compatible	Not compatible
4.1.X	Compatible	Compatible	Compatible	Not compatible
4.2.X	Compatible	Compatible	Compatible	Not compatible
5.0.X	Not compatible	Not compatible	Not compatible	Compatible

Table 2: ELinOS P4 Bus drivers compatibility matrix

The following table describes the compatibility between PikeOS and ELinOS version using PikeOS P4 Bus drivers. Theses drivers could be found in PikeOS directory in folder share/hwvirt-linux/pikeos.

ELinOS / **PikeOS**	4.0.X	4.1.X	4.2.X	5.0.X
6.0.X	Compatible	Compatible	Compatible	Not compatible
6.1.X	Compatible	Compatible	Compatible	Not compatible
6.2.X	Compatible	Compatible	Compatible	Not compatible
7.0.X	Not compatible	Not compatible	Not compatible	Compatible

Table 3: PikeOS P4 Bus drivers compatibility matrix

Note:
PikeOS P4 Bus drivers (5.0.X) supports Linux kernel version for arm architecture from 3.4 up to 4.20
and supports Linux kernel version for arm64 architecture from 3.16 up to 4.20.

4.1.2 P4 Bus operations

A P4 Bus operation is defined by the following structure:

```
typedef struct p4bus_operation_str {
/* status of the p4bus operation */
volatile uint32_t status;
/* return code of the operation */
uint32_t retcode;
/* operation ID */
uint16_t type;
/* operation flags */
uint16_t flags;
/* file descriptor used in a file operation */
uint16_t filedesc;
/* device id */
uint16_t devid;
/* Intermediate physical address */
uint64_t addr;
/* Size of the area pointed by addr */
uint64_t size;
/* spare parameter 1 */
uint64_t param1;
/* spare parameter 2 */
uint64_t param2;
```

```
/* private parameter to be used by the guest */
uint64_t priv_guest;
}  __attribute__((packed)) p4bus_operation_t;
```

The “flags” field is used to indicate a specific status regarding the current operation. It can be set with the following values:

Value	Description
OP_FLAGS_SLEEP	This value is set by the manager and indicates to the guest that it will sleep because all operations present in the IORING have been executed. If a new operation is pushed by the guest, the guest must signal the manager.
OP_FLAGS_SIGNAL	This value is set by the guest to indicate to the manager that it will to generate an IRQ once the operation has been executed.
OP_FLAGS_POLL	This value is set by the guest to indicate to the manager that no IRQ generation is required. The guest is just polling on the operation status.

Table 4: P4 Bus operation flags

The “status” field is the base of the P4 Bus operation machine-state used by the P4 Bus drivers. It can have the following value:

Value	Description
P4BUS_OPERATION_FREE	The operation is available and can be used by the guest.
P4BUS_OPERATION_READY	The operation has been configured by the guest and will be executed by the manager.
P4BUS_OPERATION_DONE	The operation has been executed by the manager. The operation results are available for the guest.

Table 5: P4 Bus operation status

4.1.3 P4 Bus Operation Types

The P4 Bus defines some standard operation types together with the corresponding attributes:

Type	Parameter 1	Parameter 2	Buffer address	Buffer size	Return value	File descriptor
Open	Open flags	-	-	-	-	(1)
Close	-	-	-	-	-	(1)
Read	Offset	0 for a read, not 0 for a read_at (2)	Read destination buffer	Buffer size	Size actually read	(1)
Write	Offset	0 for write, not 0 for a write_at (2)	Write data buffer	Buffer size	Size actually written	(1)
IOCTL	Command	size in / size out	Data buffer	Buffer size	-	(1)
MMAP	Offset	Mapping flags	NULL	Size to map	Address of the mapping	(1)
Stat	-	-	Address to a buffer (needs to be at least 64 Bytes in size)	Buffer size	Size written in buffer	(1)
Lseek	Offset	Origin	-	-	New position	(1)

Table 6: P4 Bus Operation

(1) This field is used to identify the current instance for the operation concerned. It allows the management of several OPEN/CLOSE on the same device. Currently it is only used by the VMCHAR driver.

(2) For read and write, the parameter 2 is used to switch between sequential operations (using vm_read or vm_write) and at offset operations (using vm_read_at or vm_write_at). When the parameter 2 is 0, the offset parameter is not used.

All addresses contained in the messages are physical addresses from the guest point of view. The addresses used in the protocol are IPA.

4.1.4 P4 Bus ioring

P4 Bus protocol provides also an API for **asynchronous** operations. These operations are based on the IORING concept which is defined in the **"p4bus-ioring.h"** file.

```
typedef struct p4bus_ioring_s {
/* index of the FREE entry in the ioring */
uint32_t curr_free;
/* some padding for 64bit systems */
uint32_t unused1;
/* index of the DONE entry in the ioring */
uint32_t curr_done;
/* some padding for 64bit systems */
uint32_t unused2;
/* p4bus operations IORING */
p4bus_operation_t operations[P4BUS_IORING_DEPTH];
}  __attribute__((packed)) p4bus_ioring_t;
```

From a basic point of view, an IORING is an array of operations with indexes. This indexes are managed by the IORING API functions and must not be modified outside of it.

- The curr_free index is pointing to the current FREE operation which can be used by a p4bus driver to push a NEW operation.

- The curr_done index is pointing to the current DONE operation. This status indicates that the operation has been processed by the manager and the results of the operation can be retrieved.

A P4 Bus driver using the IORING API must declare the number of p4bus_ioring_t variables it wants to use and follows these steps:

- Initialize the ioring using “p4bus_ioring_init”: This will fill the ioring structure with the initial values.
- Call the P4 Bus command “P4BUS_COMMAND_CREATE_IORING”: This command will inform the manager to attach an operation thread to the given ioring.

Once done, the P4 Bus driver is allowed to push P4 Bus operations in the ioring.

4.1.4.1 Push a new operation in the ioring

The following steps are given as example:

- Get an available operation: The **p4bus_ioring_get_free** function will return a P4 Bus operation if available.
- Configure the operation: The driver must set the operation parameters depending of it needs. The “flags” field must be set as explained in the section 4.1.2.
- Push the operation in the ioring: Call the **p4bus_ioring_push_ready** function to set the P4 Bus operation as ready.
- Inform the manager: The P4 Bus command **“P4BUS_COMMAND_SIGNAL_IORING”** will inform the manager to wakeup the operation thread handling the given ioring ID. The operation thread will execute all the operations with the status set to “P4BUS_OPERATION_READY” and for each of them, if the “flags” value is set to “OP_FLAGS_SIGNAL”, it will raise an interrupt. This command is not mandatory for each P4 Bus operation pushed in the ioring. If the driver has several P4 Bus operations to push, it can use this command after the last one.

Note: Once the operation thread has executed all the P4 Bus operations, it will sleep and will wait to be signaled with the P4 Bus command **“P4BUS_COMMAND_SIGNAL_IORING”**. Before that, it informs the guest that is going to sleep. This information is set in the last executed operation in the “flags” field with the specific value “OP_FLAGS_SLEEP”.

4.1.4.2 Retrieve ioring operation results

For each P4 Bus devices declared in the guest, the manager attaches an interrupt. This interrupt is used to signal the guest for each end of operation containing the “OP_FLAGS_SIGNAL” value in the “flags” field (see the section 4.2.

The following steps are given as example and take as prerequisite that the P4 Bus driver has declared an interrupt handler attached to P4 Bus device interrupt and each operations use the “OP_FLAGS_SIGNAL” flags value.

- In the interrupt handler, for all p4bus_ioring_t variables declared in the driver, use the **p4bus_ioring_get_done** function to retrieve the done operations.
- Check the results of the operation.
- Release the operation by using the **p4bus_ioring_push_free** function.

Note: If the interrupt feature is not used and the “flags” value is not set to “OP_FLAGS_SIGNAL”, the driver must loop on the operation status. Once the status is set to “P4BUS_OPERATION_DONE” by the manager, the results can be checked and used by the driver.

4.2 P4 Bus Devices

The P4 Bus is defining devices. Each device, from the bus point of view, is one host device connected to one guest device.

P4 Bus devices have several attributes:

- **Host type:** The type of the device on the host side. Common types available are vmcprintf, vmfile, vmqport or vmsport.
- **Host name:** The name of the device on the host side. These attribute values depend on the host type and configuration.
- **Guest type:** The type of the device on the guest side. Available types may depend on the type of guest actually in use.
- **Guest name:** The name of the device on the guest side. The attribute value depends on the guest type of the device.
- **Mask of possible operation:** A 32-bit value used as a bitmask to detect the operations possible on the device. This is automatically set by the manager depending on the host type and name.
- **Size:** An automatically set value depending on the host type that can be used by the guest to dimension its buffers.
- **Interrupt:** An automatically allocated interrupt. The interrupt number is chosen by the manager during boot time. The manager checks for an interrupt number unused by the guest. This interrupt number is used to signal the guest for each end of operation on the P4 Bus device.

Note: During boot time, the host is scanning all interrupts forwarded to the guest (Interrupts attached to a directIO device or a custom directIO device). Then it will allocate an unused (non forwarded) interrupt numbers to a P4 Bus device and set it as forwarded. If a Linux driver is using a specific interrupt which is not configured to be accessible to the guest (by a DirectIO device or a custom directIO device), then, this interrupt will be identify as FREE by the host and can be allocated to a P4 Bus device. This could result to a conflict.

On the configuration level, devices are created by the integrator on the P4 Bus by adding - for each of them - a P4 Bus device component. P4 Bus components are installed in the PIKEOS_POOL into the “virtualization/p4bus” directory. For more information on the P4 Bus device components configuration, please refer to the section 3.7.

The following chapters are listing the available host and guest device types.

4.2.1 PikeOS Host Types

PikeOS host provides several types of devices implemented in the manager which are listed in the following paragraphs.

4.2.1.1 vmfile

This is used to access any internal (e.g. *shm* or *rfs*) or external file provider (e.g. *Ethernet* or *MUXA*). This driver is able to manage 128 file descriptors per device which means that a guest can open 128 times the same device (if it is managed by the file provider in the host side).

Host Type	vmfile
PikeOS File	Complete path of the file to be accessed. For example “muxa:guest”, “eth0:0” or “rfs:mydir/myfile”.
Operations	open, close, read, write, ioctl, mmap, stat, lseek
Type ID	0
Stat content	Standard (no custom values)

Table 7: vmfile Operations

Note: If the option “Use a File Provider” is not checked, access rights of the File Provider have to be granted to the guest partition which will use the vmfile P4 Bus device. The file access rights can be set with “File access mode” parameter.

4.2.1.2 vmqport

This is used to access one or two queuing ports (one source together with one destination port of equivalent characteristic). This driver does not manage multiple file descriptors for the same device.

Host Type	vmqport
Source Port	Name of the queuing port or the name of the transmit port. Set an empty string if only a receive port is used
Destination Port	Name of the queuing port or the name of the receive port. Set an empty string if only a transmit port is used
Operations	read, read non blocking, write, write non blocking, stat
Type ID	1
Stat content	The qport total size (max_nb_messages x max_msg_size), the block size (max_msg_size), the maximum number of messages which can be read from the port until it is empty, the maximum number of messages which can be written to the port until it is full or the qport direction.

Table 8: vmqport Operations

The stat function returns the following information. All of these are accessible using a specific offset from the beginning of the buffer.

Field	Description	Content	Offset (bytes)	Size (bytes)
Host type	Unused field (present for backward compatibility)	0	0	4
Total size	Maximum size of the channel	max_nb_messages x max_msg_size	4	4
Block size	Maximum size of a message	max_msg_size	8	4
Src max messages	The maximum number of messages which can be written to the port until it is full	-	12	4
Dst max messages	The maximum number of messages which can be read from the port until it is empty	-	16	4
Channel directions	Directions available by the channel	0x1 if TX is available, 0x2 if RX is available, 0x3 if RX and TX are available	20	4

Table 9: vmqport stat Information

Note: Each READ operation on a vmqport must be done with a buffer size equal to the PikeOS queuing port size.

4.2.1.3 vmsport

This is used to access one or two sampling ports (one source together with one destination port of equivalent characteristic). This driver does not manage multiple file descriptors for the same device.

Host Type	vmsport
Source Port	Name of the sampling port or the name of the transmit port. Set an empty string if only a receive port is used
Destination Port	Name of the sampling port or the name of the receive port. Set an empty string if only a transmit port is used
Operations	read, read non blocking, write, write non blocking, stat (used to retrieve the validity of last message)
Type ID	2
Stat content	The sport total size (max_msg_size), the block size (max_msg_size), the age limit for valid messages on the destination port, the age limit for valid messages on the source port, validity status of the most recently read message or the channel available directions.

Table 10: vmsport Operations

The stat function returns the following information. All of these are accessible using a specific offset from the beginning of the buffer.

Field	Description	Content	Offset (bytes)	Size (bytes)
Host type	Unused field (present for backward compatibility)	0	0	4
Total size	Maximum size of the channel	max_msg_size	4	4
Block size	Maximum size of a message	max_msg_size	8	4
Src refresh period	The age limit for valid messages on the source port (in ns)	-	12	8
Dst refresh period	The age limit for valid messages on the destination port (in ns)	-	20	8
Dst last message validity	validity status of the most recently read message	-	28	4
Src last message validity	validity status of the most recently read message	-	32	4
Channel directions	Directions available by the channel	0x1 if TX is available, 0x2 if RX is available, 0x3 if RX and TX are available	36	4

Table 11: vmsport stat Information

Note: An IOCTL can be use to setup the Refresh Rate parameter of a receive sampling port:
See vmchar paragraph below (P4BUS_IOCTL_PIKEOS).

4.2.1.4 vmcprintf

This is used to access the PikeOS console.

Host Type	vmcprintf
Host Name	Ignored.
Operations	write
Type ID	3

Table 12: vmcprintf Operations

4.2.1.5 vmapi

This is used to access PikeOS functions and information.

Host Type	vmapi
Host Name	One of the supported functions (see next table).
Operations	read or/and write depending on the function
Type ID	4

Table 13: vmapi Operations

This specific type provides several sub-interfaces which can be called by writing a specific command to the vmapi device:

command	Argument	Function provided
reboot-partition	Partition ID	Reboot the partition (with the given ID) on the host system.
reboot-target	-	Reboot the complete system. No argument is required.
stop-partition	Partition ID	Halt the partition (with the given ID) on the system.
stop-target	-	Stop the complete system. No argument is required.
set-time-sched	Scheduling scheme name	Change the time partition scheme.

Table 14: vmapi Sub-Interfaces

Note: On a Linux guest, CONFIG_VMM_VMAPI must be defined and CONFIG_SYSFS and/or CONFIG_PROCFS must be set. This will create the following files:

- **PROCFS:** /proc/vmapi.
- **SYSFS:** /sys/vmapi/command.

The VMAPI features are directly available by writing the command and the argument (if required) to these files.

4.2.1.6 vmnull

This is used for disposing of unwanted output streams or data.

Host Type	vmnull
Host Name	Ignored.
Operations	write
Type ID	5

Table 15: vmnull Operations

4.2.2 Linux Guest Types

Linux guest device types are defined to provide the most common generic device types that are possible to implement under Linux with a design allowing to connect almost any PikeOS host types to any Linux guest types (some combinations are not possible or simply are impractical, such as connecting vmnet to vmcprintf).

4.2.2.1 vmchar

This type allows the access of the P4 Bus device through a Linux character interface. This driver allows having several accesses to the same host device. The current limit is set to 128 file descriptors per device.

Guest Type	vmchar
Guest Name	When udev is used, this parameter can be retrieved to create a device using this name. With the default configuration, the devices "/dev/vmcharX" with X being the device number, "/dev/vmchar/*GUESTNAME*" will be created.
Operations	open, read, write, ioctl, lseek, mmap, close (depending on the capabilities of the host type).

Table 16: vmchar Driver

The IOCTL provides 2 different commands:

- P4BUS_IOCTL_PIKEOS: This is used to call directly the IOCTL command of a file provider supporting it. The buffer passed to the IOCTL must be a 144 bytes structure containing:

Name	Size	Comment
Command	4 bytes	The IOCTL command number
In-Size	4 bytes	The input data size
Out-Size	4 bytes	The output data size
Return	4 bytes	The P4 IOCTL return code (P4_e_t)
Data	128 bytes	The input/output buffer

Table 17: vmchar IOCTL

Guest applications that like to make use of the IOCTL need to manually define the following IOCTL type themselves:

```
#include <linux/ioctl.h>
```

```
/* NATIVE IOCTL */
#define PIKEOS_IOCTL_DATA_SIZE      128u
#define PIKEOS_IOCTL_STRUCT_SIZE    (PIKEOS_IOCTL_DATA_SIZE + (4 * 4))
#define PIKEOS_IOCTL_CMD_FIELD      0u
#define PIKEOS_IOCTL_SIZE_IN_FIELD  1u
#define PIKEOS_IOCTL_SIZE_OUT_FIELD 2u
#define PIKEOS_IOCTL_RETURN_FIELD   3u
#define PIKEOS_IOCTL_DATA_FIELD     4u

/* PIKEOS IOCTL CMD */
#define P4BUS_IOCTL_NATIVE          0x12u

typedef struct p4bus_ioctl_s {
    uint32_t fields[PIKEOS_IOCTL_STRUCT_SIZE];
} p4bus_ioctl_t __attribute__((aligned(4)));

#define IOC_MAGIC       55
#define IOCTL_NATIVE    _IOWR(IOC_MAGIC, P4BUS_IOCTL_NATIVE, struct p4bus_ioctl_s)
```

Note: IOCTL definitions can be included by adding p4bus-vmchar.h file:
#include <p4bus-vmchar.h>
This header file can be found in share/hwvirt-linux/pikeos PikeOS directory
or in drivers/virt/pikeos ELinOS directory.

Command number to set refresh rate for (receive) sampling port (vmsport):
#define P4BUS_IOCTL_SET_REFRESH_RATE 0x1
The input data size for this command is 0x8 and the output data size is 0x0.

Example to set value at 500000000:
ioctl_str.fields[PIKEOS_IOCTL_CMD_FIELD] = P4BUS_IOCTL_SET_REFRESH_RATE;
ioctl_str.fields[PIKEOS_IOCTL_SIZE_IN_FIELD] = 0x8;
ioctl_str.fields[PIKEOS_IOCTL_SIZE_OUT_FIELD] = 0;
ioctl_str.fields[PIKEOS_IOCTL_DATA_FIELD] = (500000000 & 0xffffffff);
ioctl_str.fields[(PIKEOS_IOCTL_DATA_FIELD + 1)] = (500000000 » 32) & 0xffffffff;

- P4BUS_IOCTL_STAT: This is used to call P4 Bus stat on a device supporting it. The buffer passed to the IOCTL must be a 64 bytes structure and will contain in return:

Name	Size	Comment
TotalSize	8 bytes	The device total size
BlockSize	8 bytes	The device block size
Misc	48 bytes	Custom data depending on the host type

Table 18: vmchar IOCTL STAT

Guest applications that like to make use of the IOCTL need to manually define the following IOCTL type themselves:

```
#include <linux/ioctl.h>

/* STAT OVER IOCTL */
#define STAT_MAX_UINT64_FIELDS   8u
```

```
#define PIKEOS_STAT_STRUCT_SIZE  (STAT_MAX_UINT64_FIELDS * 8)
#define STAT_TOTAL_SIZE_FIELD    0u
#define STAT_BLK_SIZE_FIELD      1u

/* PIKEOS IOCTL CMD */
#define P4BUS_IOCTL_STAT         0x11u

typedef struct p4bus_stat_s {
    uint64_t fields[STAT_MAX_UINT64_FIELDS];
} p4bus_stat_t __attribute__((aligned(8)));

#define IOC_MAGIC       55
#define IOCTL_STAT      _IOWR(IOC_MAGIC, P4BUS_IOCTL_STAT, struct p4bus_stat_s)
```

Note: STAT definitions can be included by adding p4bus-vmchar.h file:
#include <p4bus-vmchar.h>
This header file can be found in share/hwvirt-linux/pikeos PikeOS directory
or in drivers/virt/pikeos ELinOS directory.

The mmap support allows the mapping of some external resources directly. As Linux has no “execute” flag when requesting to do MMAP, it is not possible to map an area to execute code in, into a Linux application.

Note: For a vmchar device connected to a vmqport, each READ operation on this device must be done with a buffer size equal to the PikeOS queuing port declared in the vmqport device. If this rule is not applied, the following error is raised by the vmchar driver: “p4bus_vmchar_read Error READ operation returns an error: 0x2000b”.

4.2.2.2 vmtty

This type allows the access of the P4 Bus devices through a Linux tty interface. This also supports the connections to the Linux kernel console over a P4 Bus device, if the corresponding kernel configuration is set. This can be done by setting “console=vmttyX” to the Linux kernel command line.

Guest Type	vmtty
Guest Name	When udev is used, this parameter can be retrieved to create a device using this name. With the default configuration, the devices “/dev/vmttyX” with X being the device number and “/dev/vmtty/vmttyX” will be created.
Operations	Standard tty behavior as for a serial line is supported. Setting serial line speed or flow control has no effect.

Table 19: vmtty Driver

Note: Below here are two examples of possible console argument:
1) console=vmtty0

log result:
P4BUS_vmtty: VMTTY 'vmtty0' alias is vmtty0

Note:
If udev is not used, device to use in "/dev" is '/dev/vmtty0"

2) console=vmconsole

log result:
P4BUS_vmtty: VMTTY 'vmconsole' alias is vmtty0

Note:
If udev is not used, device to use in "/dev" is '/dev/vmtty0"

4.2.2.3 vmnet

This type allows the access of the P4 Bus devices through a Linux Ethernet interface. This is made to be connected to a host driver having READ and WRITE operations (vmqport, vmfile). The MAC address attributed to the interface is obtained either dynamically according to the host driver (for example: veth_fp via the IOCTL call), or statically. In the latter case, the MAC address is built as follows.

Offset (bytes)	Size (bytes)	Data
0	1	8 lower bits from the MAC_ADDR_PREFIX
1	1	8 higher bits from the MAC_ADDR_PREFIX
2	3	24 lower bits from the guest_id
5	1	0x2 = unicast and locally administrated

Table 20: vmnet MAC Address

"MAC_ADDR_PREFIX" is set to 1 by default but can be changed in the kernel configuration [Device Driver/Virtualization drivers/PikeOS Support/PikeOS Virtualization Bus/Network devices on P4 Bus/]. The "guest_id" is automatically attributed by the P4 Bus and guaranteed the unicity of the couple "host_type/guest_type".

Guest Type	vmnet
Guest Name	This parameter is only used to inform the user on which "eth name" this device is registered. This information is printed during the P4 Bus initialization.
Operations	Standard eth behavior as for an Ethernet interface is supported.

Table 21: vmnet Driver

4.2.2.4 vmblock

This type allows the access of the P4 Bus devices through a Linux block device interface, so file provider or share memory can be used as block interface.

Guest Type	vmblock
Guest Name	When udev is used, this parameter can be retrieved to create a device using this name. With the default configuration, the devices “/dev/*GUESTNAME*” will be created.
Operations	Standard block device behavior is supported.

Table 22: vmblock Driver

4.2.3 PikeOS Guest Types

PikeOS guest device types are defined to provide an access to every PikeOS host type.

4.2.3.1 vmconsole

This type allows the connection of the PikeOS guest console to a host device (for example: vmcprintf in order to redirect the guest output on the PikeOS host console).

Guest Type	vmconsole
Guest Name	Ignored.
Operations	Standard output console (unidirectional) behavior.

Table 23: vmconsole Driver

4.2.3.2 vmfp

This type allows having a generic access to a P4 Bus device through a file interface. This type is provided by a Kernel Driver component named “Kernel Driver P4BUS VMFP” and has to be included to your PikeOS guest integration project (see the section 8.1.3). The driver is also included in the Hardware Virtualization BSP as a kdev module.

Guest Type	vmfp
Guest Name	This parameter will be used to add a reference through the vmfp file provider at vmfp:“Guest name”. For example, if it’s equal to “myFile”, the device will be accessible through the “vmfp:myFile” path in application side.
Operations	vm_open, vm_read, vm_read_at, vm_write, vm_write_at, vm_stat, vm_lseek, vm_fstat, vm_close, vm_map, vm_ioctl.

Table 24: vmfp Driver

4.3 P4 Bus Configuration

To add a P4 Bus device you must to add the appropriate “p4bus device” component depending of your Guest Type (see the section 4.2).

5 External Exception Handler

The functionalities of the PikeOS hardware virtualization can be extended to fit particular needs using **External Exception Handlers**. Those handlers are running in the PikeOS kernel (as kernel drivers or directly in a psp) and can handle some guest exceptions.

An **External Exception Handler** can handle any exception that would normally be handled by the hardware virtualization manager.

5.1 Design

5.1.1 Handler registering

An external exception handler is defined by implementing the *p4hwvirt_exception_handler_init_t*, *p4hwvirt_exception_handler_handle_t* and *p4hwvirt_exception_handler_exit_t* callback functions (See '14.6.1 HWVIRT KDEV Host interface' for a definition of these callbacks) defining:

- **An Init Function:** This function will be called when a guest is initialized.
- **An Handler Function:** This function will be called when the exception, for which the handler is registered, is raised by a guest.
- **An Exit Function:** This function will be called when a guest is destroyed.

The handler must be registered for a specific exception (See 14.2.2 for a list of exceptions and their definition) inside the hypervisor using the function **p4hwvirt_register_exception_handler** (See 14.6.2.1 for more information on this function).

This must be done once when the system is starting. In case of an handler implemented in a PSP, this should be done after hardware virtualization initialization is called on core 0. In case of a kernel driver, this should be done at the gate initialization.

Note: It must be called after P4_BOOT_STAGE_INIT_PROV and before P4_BOOT_STAGE_COMPLETED, if it is used before or within P4_BOOT_STAGE_INIT_PROV it will return P4HWVIRT_E_HYP_INIT_STATE as we can't control the order of kernel drivers init_prov functions execution.

Several handlers can be registered for the same exception.

5.1.2 Handler Guest Init

The **init** callback is called during a guest initialization and before the guest is started.

An **External Exception Handler** can filter which guests it wants to get exception for by returning *P4_E_MISMATCH* in the **init** callback for the guest it does not want to handle.

The **init** callback is not mandatory. If there is not **init** callback registered, it is considered that the exception handler will handle all guests.

Note: The guest identifier passed to all callbacks is in fact the Partition ID of the Manager of the guest.

The *priv* pointer passed to the **init** callback can be filled with a P4_address_t value that will be passed back to the **handler** and **exit** callbacks for this guest.

The function **p4hwvirt_guest_alloc** (See 14.6.2.4 for more information) can be used to allocate some memory for this guest. This memory will be lost when the guest is destroyed and must be reallocated and reinitialized each time the **init** callback is called for a guest.

5.1.3 Handler Exception Handling

The **handler** callback is called every time the guest is raising the exception the handler was registered for.

The handler can do 3 different things with an exception:

- **Handle the exception:** In this case the exception must be properly filled (for example setting the *IO value* field for an IO exception) and the handler must return *P4_E_OK*. The guest will continu its execution without calling the manager for the exception.
- **Ignore the exception:** In this case there could be an other exception handler that could handle the exception and the handler must return *P4_E_MISMATCH* to let other handlers try to handle the exception. If no handler is handling it, the exception will be finally passed back to the manager.
- **Need an action from the manager:** In case of an handler simulating a firmware, some specific actions like starting a secondary core or stopping the guest. In this case the exception structure must be modified to generate the corresponding exception to the manager and the handler must then return *P4_E_TRUNC*.
- **Generate an error and stop the guest:** In this case the handler must modify the exception type to *P4HWVIRT_EXCEPT_UNSUPP* and return an error code. If the exception type is not modified the manager will try to handle the exception normally.

When the **handler** callback is called, the handler is getting the exception information through the **exception** argument but also the complete guest context through the **context** argument.

This context is directly the internal context of the hypervisor and can be modified directly when needed. The exception structure *run_flags* is only used to check if the PC must be advanced or not, the *Update Context* bit is not used here.

For some exceptions (IO, CP15, MSR for example), the proper register is set with the exception value before restarting the guest for *READ* exceptions.

5.1.4 Handler Guest exit

The **exit** callback is called when the guest is destroyed which is when the manager partition is stopped or restarted.

The **exit** callback is not mandatory.

Note: After **exit** has been called for a specific guest, an **External Exception Handler** should not try to access any memory allocated using **p4hwvirt_guest_alloc**, thus the **exit** should make sure any interrupt handler or asynchronous handler for this guest is not called anymore

5.1.5 Asynchronous call

An **External Exception Handler** can use the function **p4hwvirt_guest_gen_irq** (See 14.6.2.2 for more information) to generate virtual interrupt to a guest asynchronously (for example from an interrupt handler in a PSP or a kernel driver).

This is usually the case when an **External Exception Handler** is simulating some hardware.

5.2 Examples

The PikeOS installation contains several **External Exception Handlers** examples provided as Kernel Driver Project examples.

5.2.1 SMC Bypass

This **External Exception Handler** is handling guest SMC exceptions.

The SMC instruction is forbidden to guests as it is used to communicate with the board firmware usually and this part is simulated by the manager so that a guest only has access to the wanted ressources and cannot get a control of the full target.

Normally the guest should use the HVC instruction instead of the SMC instruction when running as guest to communicate with the firmware, which is normally easy to configure for a Linux or a PikeOS guest.

This driver is doing 2 different things:

- **Forward PSCI request to the manager**

  If for some reason a guest cannot be modified to use the HVC instruction, this handler will get the SMC exception and, in case of a standard PSCI request, will transform the exception in an HVC exception and forward the request to the manager.

  This is enough to have the guest thinking it is discussing with the firmware to boot secondary cores or stop or reset the target.

- **Forward non PSCI request to the firmware**

  On some boards, the PSCI firmware has some extra capacities (encryption, clock management, etc) which could be needed by a complex guest. This handler will let those request passthrough to the firmware and will forward the answers to the guest (standard PSCI requests will continu to be handled by the manager).

Note: This driver is provided as example. On a real system forwarding request directly to the firmware might break the partitionning and some filtering would need to be implemented in the handler

5.2.2 Virtio Memory

This **External Exception Handler** is handling IO exception and is simulating a memory for access between 0x1000 and 0x2000 physical addresses.

For this the driver is:

- Registering itself for IO exceptions during init.
- Allocating itself one page of memory that will be used to store guest write values.
- Filtering IO exceptions to handle only IO access to the right address.
- In case of write access, writting the value in the guest allocated memory.
- In case of read access, returning the content in the allocated memory at the proper offset.

This example is showing how the *priv* can be used together with the guest memory allocator and also how an IO exception is encoded and can be handled.

5.2.3 mrs msr bypass

This **External Exception Handler** is handling co-processor access exception and is stubbing the accesses.

For this the driver is:

- Registering itself on CP15 32bits, CP15 64bits and MRS/MSR exceptions during init.
- Depending on the Verbosity level, log every accesses on the console.
- In case of write access, doing nothing.
- In case of read access, returning 0.

This example is showing how an **External Exception Handler** is able to handle several differents exception. It can be also used as a starting point to emulate co-processors.

6 VM error handling

Since the 5.0.2 version of PikeOS, all the errors raised by a VM have been reworked to provide a better handling/understanding of them. All the errors are now defined by an unique ID as shown in the section HWVIRT Errors in chapter 14 and are attached to a group (see the section 6.2). The VM error handling configuration is available in the **"Virtualization partition"** component from your guest.

6.1 VM error handling strategies

The VM error handling strategy defined, for a specific VM, the behaviour to have when an error occures.

- **Health Monitor handling:** Each error is attached to an error group (please see the section 6.2) on which a Health monitor action can be configured in the VM partition component. With this option, a health monitor error is injected and the Health Monitor message is printed. An action can be attached to handle the error. On the following example, we can see the Error ID - 0x1000C which is in the HWVIRT_T_HYPERVISOR_GUEST group and attached to the action "P4_HM_PAC_IDLE" by default.

```
#### HM EVENT in 23.0 "guest1"."guest1":
#### type=0x0 P4_HM_TYPE_UINT, id=0x10000
#### msg="[P4HWVIRT_E_HYP_NO_HYPMEM]: Not enough hypervisor memory"
#### lvl=0x1 P4_HM_LEVEL_PARTITION, dom=0x8 USER, code=0x0
#### pac=0x0 P4_HM_PAC_IDLE, mac=0x1 P4_HM_MAC_SHUTDOWN
#### pnotify=0x0, mnotify=0x0
```

- **P4_HM_PAC_COLD_START:** The VM is restarted in the operating mode cold start for any error raised.
- **P4_HM_PAC_IGNORE:** Errors are ignored (all errors attached to the HWVIRT_T_XXX group). If a manual configuration in the VMIT is set, the action is done else the VM will be halted if the error is fatal.
- **P4_HM_PAC_IDLE:** The VM is halted for any error raised.
- **P4_HM_PAC_WARM_START:** The VM is restarted in the operating mode warm start for any error raised.

6.2 VM error groups

All the error IDs are attached to a group allowing to define (for some of them) a Health Monitor action when the error occures.

Error Group	Description
HWVIRT_T_HYPERVISOR_GUEST	Health Monitoring actions for Hypervisor errors.
HWVIRT_T_MANAGER_CONFIG	Health Monitoring actions for Manager Configuration errors.
HWVIRT_T_MANAGER_DIRECT_IO	Health Monitoring actions for Direct IO module errors.
HWVIRT_T_MANAGER_FDT	Health Monitoring actions for FDT module errors.
HWVIRT_T_MANAGER_GUEST	Health Monitoring actions for Guest errors.
HWVIRT_T_MANAGER_VMM	Health Monitoring actions for VMM errors.
HWVIRT_T_MANAGER_P4BUS	Health Monitoring actions for P4BUS errors.

HWVIRT_T_MANAGER_CPU	Health Monitoring actions for CPU errors.
HWVIRT_T_MANAGER_GENERIC	Health Monitoring actions for generic manager errors.

Table 25: VM error groups

7 Linux as Guest

This chapter describes how to run Linux as an hardware virtualization guest.

The Linux kernel for ARM architecture uses two ways to find the boot information (devices, memory). The hardware virtualization support both of them. They have to be configured into the PikeOS process configuration (see the section 3.5.9).

- **Device Tree:** DTB support will load a DTB file at a given offset in guest memory on start-up and the manager will modify it in the same way a bootloader would do. A specific option ("Linux Boot" in the hwvirt Process component) allows deactivation of all modifications done on the original DTB.
  - Linux Kernel command line: The manager will replace the default command line by the one defined in the hwvirt Process component.
  - Memory: The manager will remove all memory nodes to add those configured for the guest (The guest RAM and all "Virtualization Memory" components defined).
  - CPUs: The manager will remove all CPU nodes to replace them according to the cpumask of the hwvirt partition. The manager will also include the PSCI callbacks addresses to allow the guest to manage the cores through the emulated PSCI.

  A parameter allows you to define the offset in the guest memory at which the DTB is actually loaded. You should modify this parameter value to make sure the offset is not outside of the guest memory and is also not overwriting other guest loaded files (i.e. if your Linux kernel's size has been changed).

Note: If you don't want any modifications (done by the manager) in your DTB and provide your own, you must have to disabled the DTB support here and add a "Virtualization File" component configured to load the DTB for a given address.

7.1 Guest Configuration

Note: For an ELinOS guest, please refer to the ELinOS Platform Manual (PikeOS Virtualization chapter).

If you want to execute a Linux as guest, you must check the following conditions:

- Compile Linux as zImage: The manager cannot extract an u-boot image and some complexity would be introduced by booting directly a vmlinux (as it will not relocate itself). The easiest way is to create a zImage Linux.
- Ram Filesystem in kernel: If you want to use a RAM file system (initial or as complete filesystem), it must be included in the kernel binary itself.
- Make sure the Linux kernel does not access anything unauthorized.

7.1.1 Kernel configuration

In the Linux kernel configuration, the following symbols are required to use the PikeOS Virtualization:

- CONFIG_OF: Device Tree and Open Firmware support.
- CONFIG_ARM_PSCI: Support for the ARM Power State Coordination Interface (PSCI).
- CONFIG_ARM_GIC: Support for the ARM GIC.

- CONFIG_ARCH_TIMER: Support for the ARM generic timer.
- CONFIG_ARM_PATCH_PHYS_VIRT: Patch physical to virtual translations at runtime.
- CONFIG_AUTO_ZRELADDR: Auto calculation of the decompressed kernel image address.

These are set automatically by the “CONFIG_PIKEOS_HWVIRT” option provided in the file “drivers/virt/pikeos/Kconfig”.

7.1.2 P4 Bus drivers in custom Linux kernel

In your PikeOS installation you will find a directory containing the P4 Bus and PikeOS Virtualization drivers in the directory “share/hwvirt-linux”.

This directory contains the following:

- pikeos: This directory contains the linux drivers for PikeOS Virtualization support.
- patches: This directory contains some patches examples to integrate the drivers in a Linux kernel.
- udev-rules: Some udev rules examples to be used with PikeOS Virtualization.
- Makefile.elinos: Makefile useable in an ELinOS project.
- README: Text file containing additionnal informations.

Note: To compile the PikeOS Virtualization drivers as modules:

- Activate Module support in your Linux.
- Copy the content of the “share/hwvirt-linux/pikeos” sub-directory in a work directory “WORK_DIR”.
- In your kernel compilation directory call the following command: “make M=WORK_DIR pikeos modules_prepare modules modules_install”.
- If you don’t want to have all drivers, edit “WORK_DIR/Makefile” and comment out the lines "CONFIG_xxx := m" for the drivers you don’t want to have.

Note: To compile the PikeOS Virtualization drivers as internal kernel drivers:

- Create a directory drivers/virt/pikeos in your Linux sources
- Copy the content of the “share/hwvirt-linux/pikeos” sub-directory in this directory.
- Edit the files drivers/virt/Makefile and drivers/virt/Kconfig to instruct linux to go in the sub-directory “pikeos”. You can find in the directory “patches” examples showing how to modify them.
- run “make menuconfig” in your kernel, go to “drivers -> virtualization -> PikeOS Support” and activate the drivers you want to use.
- build your kernel.

For each P4 Bus devices declared in your integration project, the Manager will add a node in the DTB. This node will be parsed by the Linux kernel and the device will be probed. If tou don’t want the manager to add the p4bus devices in the DTB, the symbol “CONFIG_VMM_ENUMERATE” must be defined in your Linux Kernel configuration.

Note: The drivers have been tested for kernel versions 3.8, 3.12 and 4.1 and should be possible to be compiled and used with the following kernel version:

- **ARM:** from 3.4 to 4.20
- **ARM64:** from 3.16 to 4.20

7.1.3 Add an external initrd for the guest kernel

If you want to use an external initrd with your guest kernel, you must follow the steps below:

- Add a **"Virtualization File"** component to your guest (see the section 3.6) and configure it regarding your initrd file.
- modify the **"Guest Memory"** section in the "Virtualization Process" component (see the section 3.5). Set the option "Use real physical addresses" option and set "RAM guest physical address" to the real memory address found in the DTB.
- Modify the DTB as explain in the section 7.1.4.3.

7.1.4 DTB modifications

7.1.4.1 U-boot modifications

In some specific cases the u-boot provided by the board manufacturer modifies the DTB just before to boot the Linux kernel. In this case, the DTB provided with the Linux kernel could be not relevant on specific devices. To be able to retrieve the DTB patched by u-boot, please follow the steps below:

- Boot the Linux kernel natively (without PikeOS), the kernel configuration options "CONFIG_PROC_DEVICETREE" and "CONFIG_PROC_FS" must be set.
- In the prompt, use the "dtc" tool with the "fs" option to extract the DTB from the directory "/proc/device-tree" to the output format you want (DTS or DTB).
- Retrieve the extracted DTB and use it for the guest.

7.1.4.2 ARM gic

When the guest supports the DTB, the PikeOS Virtualization requires a specific synthax for the "gic" interrupt controller node. The "gic" must be configured as **interrupt-parent at the root of the device tree**:

```
/ {
...
interrupt-parent = <&gic>;
...
};
```

The "gic" must be declared as follow:

```
gic: interrupt-controller@f7ff0000 {
compatible = "arm,gic-400";
#interrupt-cells = <3>;
#address-cells = <0>;
interrupt-controller;
reg = <0x0 0xf7ff1000 0 0x10000>,
      <0x0 0xf7ff2000 0 0x2000>,
      <0x0 0xf7ff4000 0 0x2000>,
      <0x0 0xf7ff6000 0 0x2000>;
interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(8) | IRQ_TYPE_LEVEL_HIGH)>;
};
```

Note: Addresses and interrupts are given as example.

The "compatible" field must contain one of the following string:

- “arm,gic-400”.
- “arm,cortex-a15-gic”.
- “arm,cortex-a9-gic”.
- “arm,cortex-a7-gic”.

7.1.4.3 Initrd

If an external initrd is added to the guest, the following changes must be done in the DTB:

```
chosen {
...
...
linux,initrd-start = <0x94000000>;
linux,initrd-end = <0x94151400>;
...
...
};
```

- **linux,initrd-start**: must contains the initrd start address, please be compliant with the “RAM guest physical address” value from the Virtualization Process component and the “Load offset” value from the Virtualization File component correspondig to the initrd.
- **linux,initrd-end**: must be set to “linux,initrd-start + initrd-size”.

7.1.5 Udev Facilities

If some P4 Bus devices have been created, they will be available by default through the following path: “/dev/guesttypedeviceID” For example “/dev/vmchar0” represents the first vmchar device defined on the P4 Bus.

All P4 Bus devices are characterized by three attributes:

- guesttype
- guestname

These attributes can be used by custom udev rules to create symbolic links for P4 Bus devices. The following rule is the default one used by ELinOS (10-p4bus.rules):

```
KERNEL=="vmchar*", SYMLINK+="%s{guesttype}/%s{guestname}"
KERNEL=="vmchar*", SYMLINK+="vmchar/%k"
KERNEL=="vmtty*", SYMLINK+="%s{guesttype}/%s{guestname}"
KERNEL=="vmtty*", SYMLINK+="vmtty/%k"
KERNEL=="vmblock*", SYMLINK+="%s{guesttype}/%s{guestname}"
KERNEL=="vmblock*", SYMLINK+="vmblock/%k"
```

For vmnet devices, it is possible to add some specific udev rules to rename the interfaces (70-persistent-net.rules):

```
ACTION=="add", SUBSYSTEM=="net", ATTR{guesttype}=="vmnet", ATTR{guestname}=="ethfp", KERNELS=="eth*",
ACTION=="add", SUBSYSTEM=="net", ATTR{guesttype}=="vmnet", ATTR{guestname}=="ethqp", KERNELS=="eth*",
ACTION=="add", SUBSYSTEM=="net", ATTR{guesttype}=="vmnet", ATTR{guestname}=="ethvmfprov", KERNELS=="e
```

To request device events from the kernel, this command can be used with udevadm tool:

```
udevadm trigger --subsystem-match=net --action=add
```

The events are displayed:

```
vmnet 1.ethfp ethfp: renamed from eth0
vmnet 2.ethqp ethqp: renamed from eth1
vmnet 3.ethvmfprov ethvmfprov: renamed from eth2
```

The result can be also checked with ifconfig tool (ifconfig -a):

```
ethfp     Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:7


ethqp     Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:8


ethvmfprov Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:9
```

7.2 Host Configuration

In the host integration project, you must add a hardware virtualization guest and select the type “linux”.

Linux can be started using its real RAM address, which prevents troubles when using Direct IO and devices having DMA support.

Depending on your needs and the capacity of the Linux you actually want to run, you may want to activate Direct IO and add some P4 Bus linux devices, if you need to use some PikeOS functions from Linux.

Most common P4 Bus devices that you may need to define:

- Console over a MUXA channel
  - Host Type: vmfile
  - PikeOS File: muxa:myguest
  - Guest Type: vmtty
  - Guest Name: vmtty0
- Network using a PikeOS network driver channel

  - Host Type: vmfile
  - PikeOS File: eth0:1
  - Guest Type: vmnet
  - Guest Name: vmnet0

- Queuing Ports RxPort and TxPort accessible through a character device
  - Host Type: vmqport
  - Source Port: TxPort
  - Destination Port: RxPort
  - Guest Type: vmchar
  - Guest Name: vmqport0

- Shared Memory accessible through a character device
  - Host Type: vmfile
  - PikeOS File: shm:myshm
  - Guest Type: vmchar
  - Guest Name: vmshm0

Note: If the option “Use File provider” is not set to true, access rights of the File Provider have to be granted to the guest partition which will use the vmfile P4 Bus device.

Note: The devices in Linux will be numbered base on the order of the entries in the property file system of PikeOS. As the romimage is sorting alphabetically the entries, the device numbers in Linux can be deducted from the alphabetic order of the P4Bus **Device Name** attributes.

8 PikeOS as Guest

Executing PikeOS as guest means having a complete PikeOS system, with partitions and applications on those partitions, running as guest of the host's PikeOS system.

8.1 Creating The Guest System

8.1.1 Hardware Virtualization Guest PSP

To create a PikeOS Hardware Virtualization guest system, you must create an integration project and select the board named "pikeos-hwvirt". The Boot Strategy used is "Raw" and cannot be set to an other value. The PikeOS guest can use multiple cores if the host configuration allows it.

This PSP does not support Direct IO so no drivers can be used and the P4 Bus is used to communicate.

8.1.2 Console Configuration

PikeOS as guest supports 3 kinds of consoles:

- PikeOS host console as guest console: This can be configured by selecting the console number "1" for ARMV7 guests, "UART1" for ARMV8, on your PikeOS guest.
- P4 Bus device as console: The P4 Bus device needs to support the "write" operation and the guest type of the device must be "vmconsole". The PikeOS console number must be "2 + device number" for ARMV7 guests, "UART2 + device number" for ARMV8. "device number" can be greater than 0 if you configure several devices with the guest type "vmconsole" on your host configuration.
- Standard device console (serial or other): When your guest is accessing directly (DirectIO) a console device and the proper device is granted to your guest, a standard device can be used as console.

8.1.3 Access to P4 Bus Devices

Access to P4 Bus devices is only possible through a Kernel Driver component providing the "vmfp" file prefix on your guest system.

This Kernel Driver component is named "Kernel Driver P4BUS VMFP" (installed in the "PIKEOS_POOL/kerneldriver" directory) and has to be included in your PikeOS guest integration project.

Note: The VMFP Kernel Driver component is included by default by creating an integration project using the "pikeos-hwvirt" board.

The VMFP component is seen as a file provider from the PikeOS application point of view. To allow your application to use it you need to set the option "Enable file-provider access (e.g. MUXA)" to true (in the PikeOS Process options). It will activate a dependency on a file provider which has to be resolved by the VMFP component. By this way, all access rights needed will be automatically configured (and can be overwritten in the VMIT).

This Kernel Driver provides a file interface between the P4 Bus devices and a PikeOS application running in a PikeOS partition of the guest. All P4 Bus devices configured with the guest type "vmfp", will be accessible on the guest through the file "vmfp:<guestname>" where "guestname" is the guest name configured on the host for this device.

For example if you configured a P4 Bus device with guest type "vmfp" and guest name "myfile", it will be accessible by using the file "vmfp:myfile" on your guest system.

8.2 Host Configuration

In the host's integration project, you must include a "HWVIRT PikeOS" component (see section 3).

Depending on your needs and the capacities of the PikeOS guest, you may want to add Direct IO accesses to devices and add some P4 Bus PikeOS devices.

Most common P4 Bus devices that you may need to define are:

- PikeOS Console over a MUXA channel
  - Host Type: vmfile
  - Host Name: muxa:myconsole
  - Guest Type: vmconsole
  - Guest Name: unused

  The guest must be configured to use the console number "33" to use it.
- Access to a MUXA channel
  - Host Type: vmfile
  - PikeOS File: muxa:myguest
  - Guest Type: vmfp
  - Guest Name: myMuxaChannel

  This device will be available through the "vmfp:myMuxaChannel" file interface.
- Network using a PikeOS network driver channel
  - Host Type: vmfile
  - PikeOS File: eth0:1
  - Guest Type: vmfp
  - Guest Name: ethernet

  This device will be available through the "vmfp:ethernet" file interface.
- Queuing Ports RxPort and TxPort accessible through a vmfp file interface
  - Host Type: vmqport
  - Source Port: TxPort
  - Destination Port: RxPort
  - Guest Type: vmfp
  - Guest Name: myQportChannel

  This device will be available through the "vmfp:myQportChannel" file interface.
- Shared Memory accessible through a vmfp file interface
  - Host Type: vmfile
  - PikeOS File: shm:myshm
  - Guest Type: vmfp
  - Guest Name: vmshm0

  This device will be available through the "vmfp:vmshm0" file interface.

Note: If the option "Use File provider" is not set to true, access rights of the File Provider have to be granted to the guest partition which will use the vmfile P4 Bus device.

9 Direct IO Guest

This chapters describes how to run Direct IO hardware virtualization guest.

9.1 Direct IO support in a Linux guest

The Linux kernel retrieves the hardware description from the device tree blob (DTB) (description in the section 7). For a complete DirectIO support, the native DTB must be used in the guest and the hardware registers must be mapped to the guest. This can be easily achieved with the following components provided by the PikeOS Virtualization support:

- DirectIO devices component: See the section 11 which describes the component for each supported board.
- Custom DirectIO component: See the section 3.10.

9.2 DTB modifications

In order to remove some devices from Linux configuration without changing kernel configuration (recompile), it is possible to disable some devices directly in the dtb using one of the following solutions:

- Remove unnecessary devices entries from dts file
- Set unnecessary devices *status* property to *disabled* (This solution avoids too many changes in the DTB)

Note: To ease the process depending on the Linux distribution used, we recommand to take the final DTB of your Linux, turn it back into a DTS using the *dtc* tool, modify it and finally recreate a DTB using *dtc* again.

9.3 Serial Console

PikeOS console and serial line of direct IO guest could have a conflict if the hardware used is the same.

In case of conflict, some or all of PikeOS console characters could be lost. In the worst case PikeOS console could end up in a deadlock if no characters are going out anymore (this can happen when Linux is turing the serial in DMA mode).

It should be checked that PikeOS console is not using the same hardware than Linux.

If the serial line is required by Linux you can:

- Turn off PikeOS console by selecting the *Null Console* in your PSP configuration.
- Select an other serial line for the PikeOS console.
- Disable the serial line for your guest and use a p4bus device as console.

9.4 GIC shadow registers

On systems using a GIC-400 or a GIC-500, CPU registers are also accessible with an offset of 0xf000 compared to the board documented value. Those are usually named shadow registers and are present to properly support systems with page tables with a size up to 0x10000.

PikeOS or Linux might be using the shadow registers or the standard ones. The parameter to use the PSP values for the GIC will map only the registers actually used by the PikeOS host and your guest might access the other registers.

In this case you must unselect the parameter *Use PSP register values* and set manually the DIST and CPU registers value depending on the address actually used by your guest. You can check the values used by the PSP and add/remove the 0xf000 offset to the CPU registers value (you must keep the same value for the DIST registers).

9.5 Clock / Voltage

Usually all clock and voltage registers are in the same area of registers for all IPs.

When granting access to a guest to those registers, it must be checked that the guest cannot modify the clock or the voltage of IPs which are not assigned to it. If the guest can control the voltage or the clock of something it shall not be using, it could modify gain some unwanted control on those device and affect a driver running in an other partition or in the worst case the complete system (for example if the CPU clock frequency can be modified by Linux).

All clock or voltage accessible by the guest drivers must be known by the integrator to identify which ones can be problematic for the whole system.

Please refer to the section 12 for troubleshooting.

9.6 Troubleshooting and limitations

Please refer to the section 12 for troubleshooting.

Please refer to the section 13 for limitations.

10 First Step Using Demos

10.1 Introduction

This chapter aims to guide the integrators starting with the hardware virtualization. Hardware virtualization is provided with two demos allowing the integrators to have a first look on the integration, the components and the design. The demos are based on the most common use cases:

- **hwvirt-guest-pikeos**
- **hwvirt-guest-linux**

Note: Both of them need to be used with a board supporting the hardware virtualization

10.2 hwvirt-guest-pikeos Demo

This is an integration demo and creates a project including:

- **devel group:**
  - Muxa (with Channel 1 named guest1 for guest console).
  - Monitor
  - Traceserver
- **guest1 group:**
  - guest1-part: Guest partition
  - guest1-proc: Configured for Hardware virtualized PikeOS guest and to take the guest binary directly in PIKEOS_POOL using “boot-images/simple-pikeos-pikeos-hwvirt-xxxx-raw” which is an Hello World precompiled example with "xxxx" meaning v7hf or v8hf.
  - guest1-dev1: p4bus pikeos device configured to put the guest console on the muxa channel.

10.2.1 Create a Project Based on the Demo

As with the other integration demos, you need to create a new integration project (New PikeOS Project > integration project) and select the template **hwvirt-guest-pikeos** (Demo projects > hwvirt-guest-pikeos). Then select your board supporting the hardware virtualization.

The next configuration steps are required to obtain a bootable image:

- **Configure the Console:** The PikeOS guest console (guest1-dev component) is configured as a vmfile using a File Provider dependency. This dependency needs to be satisfied with a MUXA channel. Then the MUXA needs to be configured to use a PikeOS Serial driver or a PikeOS Ethernet driver.
- **Configure the path of the hardware virtualization guest image:** Configure the path of the image by setting the “Guest binary” variable in the guest1-proc component. For a brief test, you can use one of the precompiled images in your PikeOS installation (PIKEOS_POOL/boot-images/simple-pikeos-pikeos-hwvirt-v7hf-raw for a v7hf PikeOS guest or PIKEOS_POOL/boot-images/simple-pikeos-pikeos-hwvirt-v8hf-raw for a v8hf PikeOS guest).

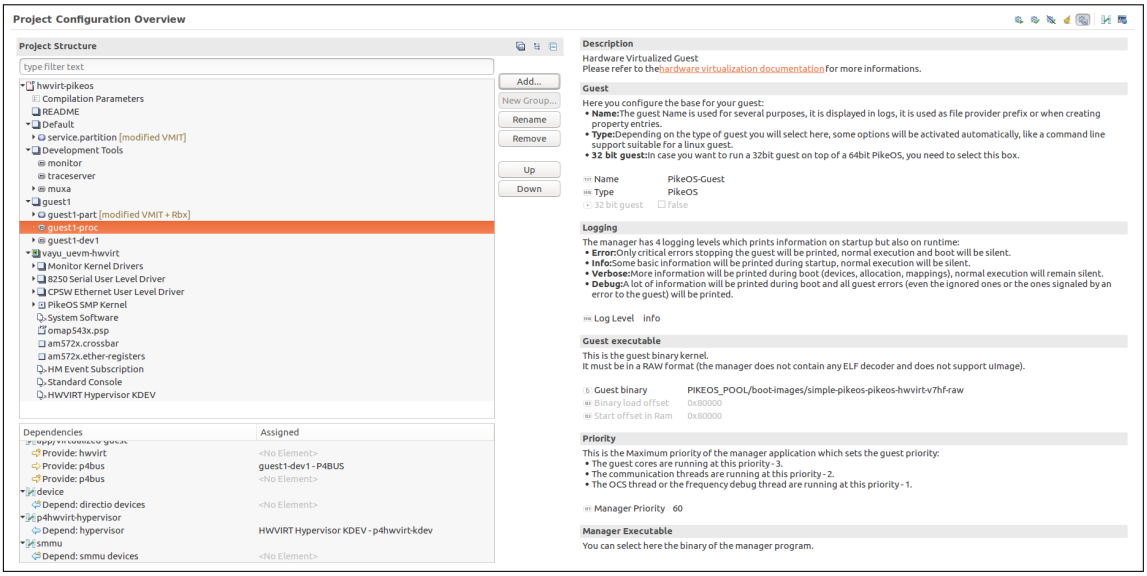

Figure 10: hwvirt-guest-pikeos Demo Overview

As shown by the previous figure, your project contains a group “guest1” containing everything needed to have your system running and a “devel” group with useful tools for the devel phase. You just need to compile it (if no modifications are needed) and boot the image on your target.

Here is an example of the demo output on the Texas Instruments Vayu board:

```
PSP booting in non-secure mode
CPU#0: Cortex A15 r2p2
PikeOS (C) Copyright SYSGO AG, Germany
ROM image build: devel-user@host-160519-16:48
Kernel build: D5.0-3289, type:  noassert tracesys smp v7 lpae [gcc]
ASP: "arm_v7hf" ARM v7, LPAE, endian: little, VFP: d0-31
PSP build: D5.0-3289
PSP: "omap543x" Texas Instruments OMAP543x (SMP-LPAE-UNTRUSTED)
Features: RETAIL TRACER-SYSCALL OPT SMP(2/32)
Configuration limits:
  respart:    255
  task:       255
  thread:     4095
  timepart:   255
  priority:   256
  kprio:      32
  interrupts: 1024
  TP windows: 256
  thr sstack: 4096 B
CPU#1: Cortex A15 r2p2
Timer: non-secure physical (irq: 30, freq: 6144000, factor: 163, reload: 61349)
P4hwvirt-kdev: loaded, 1 guests
Resource partition 0 kernel memory refill strategy: dynamic (on demand)
Time stamp counter clock: 1000000 kHz, via system call
System ticker: periodic mode, resolution 10000000 ns
Time partition switch: 10000000 ns, watchdog timeout: 10000000 ns
Time partition synchronization: default
PSP startmode: 0
```

```
PSP haltmode: 0
Free memory: 1562468 KiB
PSSW +Ext. FPs +VirtSplit +Messages (Production), Build: D5.0-1671
PIKEOS_MON: Started, version: D5.0-56
Trace Server: version: D5.0-209
<DRV INFO> eth0: PHY found: NS DP83865 (id=20005c7a)
<DRV INFO> eth0: cpsw: MAC c4:ed:ba:b6:1c:1a assigned to port 0
<DRV INFO> eth0: Registered MAC address(02:70:34:b6:1c:1a) for channel '0'
<DRV INFO> eth0: Registered MAC address(06:70:34:b6:1c:1a) for channel '1'
<DRV INFO> eth0: PHY found: NS DP83865 (id=20005c7a)
<DRV INFO> eth0: cpsw: MAC c4:ed:ba:b6:1c:1b assigned to port 1
<DRV INFO> eth0: Registered MAC address(02:70:34:b6:1c:1b) for channel '2'
<DRV INFO> eth0: Registered MAC address(06:70:34:b6:1c:1b) for channel '3'
CPSW: Provider "eth0" started, Build: D5.0-114 Production
MUXA: Version: D5.0-334
<DRV INFO> eth0: cpsw: MAC 02:70:34:b6:1c:1a assigned to port 0
<DRV INFO> eth0: cpsw: MAC 06:70:34:b6:1c:1a assigned to port 0
PikeOS-Guest(22): Manager starting (partition hwvirt_part:4, process PikeOS-Guest:22)
PikeOS-Guest(22): MANAGER: configuration
8250: Provider "ser0" started, Build: D5.0-198 Production
PikeOS-Guest(22): VMM init
PikeOS-Guest(22): MANAGER: 2 core(s) detected
PikeOS-Guest(22): Memory init
PikeOS-Guest(22): Threads init
PikeOS-Guest(22): Guest creation
PikeOS-Guest(22): Guest init
PikeOS-Guest(22):     guest memory: 0x10000000 in 1 sections
PikeOS-Guest(22): File init
PikeOS-Guest(22): Registers init
PikeOS-Guest(22): Guest core threads creation
PikeOS-Guest(22): Direct-IO init
PikeOS-Guest(22):     IOMMU enabled 0
PikeOS-Guest(22): DTB support not enabled
PikeOS-Guest(22): VMM drivers init
PikeOS-Guest(22):     VMM p4bus init
PikeOS-Guest(22):         1 p4bus devices
PikeOS-Guest(22):         64 p4bus operation threads
<DRV INFO> eth0: Link down on slave port 0
<DRV INFO> eth0: Link down on slave port 1
<DRV INFO> eth0: Link up on port 0, 1000 MBPS, full duplex
<DRV INFO> eth0: Link up on port 1, 1000 MBPS, full duplex
PikeOS-Guest(22): virtio init
PikeOS-Guest(22):     no areas, virtio disabled.
PikeOS-Guest(22): Manager Memory:
PikeOS-Guest(22):     free 0x0076a000
PikeOS-Guest(22): Hypervisor Memory:
PikeOS-Guest(22):     free 0x0000000000070000
PikeOS-Guest(22): Guest starting.
PikeOS-Guest(22): VMM-WATCHDOG: VMM_WDT_REFRESH request on a disable watchdog
```

Here is what you will see on the muxa channel:

```
PSP booting
Limiting number of interrupts to 1020, 1020 supported in hardware.
```

```
CPU#0: Cortex A15 r2p2
PikeOS (C) Copyright SYSGO AG, Germany
ROM image build: D5.0-88
Kernel build: D5.0-3289, type:  noassert tracesys smp v7 lpae [gcc]
ASP: "arm_v7hf" ARM v7, LPAE, endian: little, VFP: d0-31
PSP build: D5.0-3289
PSP: "p4hwvirt" PikeOS HWVIRT Guest (SMP-LPAE)
Features: RETAIL TRACER-SYSCALL OPT SMP(2/32)
Configuration limits:
  respart:    255
  task:       255
  thread:     4095
  timepart:   255
  priority:   256
  kprio:      32
  interrupts: 1024
  TP windows: 256
  thr sstack: 4096 B
wdt: No init_bin_config() entry point
CPU#1: Cortex A15 r2p2
Timer: virtual (irq: 27, freq: 6144000, factor: 163, reload: 61349)
hwvirt_pikeos_wdt: Provider "wdt" started, Build: D5.0-23 Production
Resource partition 0 kernel memory refill strategy: dynamic (on demand)
Time stamp counter clock: 1000000 kHz, via system call
System ticker: periodic mode, resolution 10000000 ns
Time partition switch: 10000000 ns, watchdog timeout: 10000000 ns
Time partition synchronization: default
PSP startmode: 0
PSP haltmode: 0
Free memory: 258608 KiB
PSSW +Ext. FPs +VirtSplit +Messages (Production), Build: D5.0-1671
Hello World, starting up.
Hello World, this is task 2, thread 0
Hello World, this is task 2, thread 0
Hello World, this is task 2, thread 0
```

10.3 hwvirt-guest-linux Demo

This is an integration demo and creates a project including:

- **devel group:**
  - Muxa (with Channel 1 named guest1 for guest console).
  - Monitor
  - Traceserver
- **guest1 group:**
  - guest1-part: Guest partition
  - guest1-proc: Configured for a Hardware virtualized linux guest.
  - guest1-dev1: p4bus device configured to put the linux console on a MUXA channel.
  - guest1-dev2: p4bus device configured to provide an ethernet interface to the Linux guest using a PikeOS ethernet driver.

10.3.1 Create a Project Based on the Demo

As with the other integration demos, you need to create a new integration project (New PikeOS Project > integration project) and select the template **hwvirt-guest-linux** (Demo projects > hwvirt-guest-linux). Then select your board supporting the hardware virtualization.

The next configuration steps are required to obtain a bootable image:

- **Muxa:** The “Mode” needs to be configured depending of the drivers provided by the board (Serialfp or Networkfp). Then the dependency on the port needs to be filled.
- **guest1-proc:** The paths to the guest image, the command line file and the DTB file need to be configured (see the section 3.5.9 and the section 3.5) to the file resulting from your Linux Secure IO compilation. For more information on how to create a Linux guest, please refer to the section 7.
- **guest1_dev2:** This device is preconfigured to use an Ethernet driver provided by the board. The dependency needs to be filled with the right channel provided by a PikeOS ethernet driver. If no one is available, you need to remove this device.

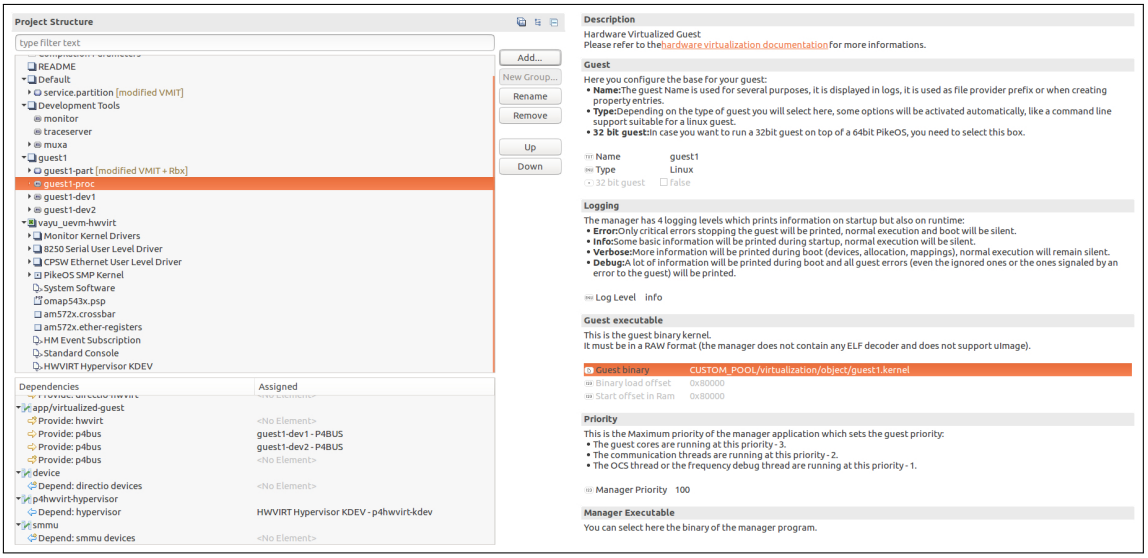

Figure 11: hwvirt-guest-linux Demo Overview

As shown by the previous figure, your project contains a group “guest1” containing everything needed to have your system running and a “devel” group with useful tools for the devel phase.

Before compiling your project, you will need a Linux Guest. To create one using ELinOS, you will have to follow the next steps:

- Create a new ELinOS project and choose the “BusyBox” example.
- In the list of available boards choose “PikeOS secure I/O ... (HWVIRT)”. You need to choose the one for your architecture.
- Compile your ELinOS project.
- Modify your PikeOS integration project to have the guest image, command line and DTB pointing to the files generated in the *boot* directory of your ELinOS project:
  - Guest image pointing to *boot/linux.kernel* of the ELinOS project.
  - Command line pointing to *boot/linux.params* of the ELinOS project.

    - DTB pointing to *boot/linux.dtb* of the ELinOS project.

- Compile your PikeOS project.

Here is an example of the demo output on the Texas Instruments Vayu board:

```
PSP booting in non-secure mode
CPU#0: Cortex A15 r2p2
PikeOS (C) Copyright SYSGO AG, Germany
ROM image build: devel-user@host-170519-14:35
Kernel build: D5.0-3292, type:  noassert tracesys smp v7 lpae [gcc]
ASP: "arm_v7hf" ARM v7, LPAE, endian: little, VFP: d0-31
PSP build: D5.0-3292
PSP: "omap543x" Texas Instruments OMAP543x (SMP-LPAE-UNTRUSTED)
Features: RETAIL TRACER-SYSCALL OPT SMP(2/32)
Configuration limits:
  respart:    255
  task:       255
  thread:     4095
  timepart:   255
  priority:   256
  kprio:      32
  interrupts: 1024
  TP windows: 256
  thr sstack: 4096 B
CPU#1: Cortex A15 r2p2
Timer: non-secure physical (irq: 30, freq: 6144000, factor: 163, reload: 61349)
P4hwvirt-kdev: loaded, 1 guests
Resource partition 0 kernel memory refill strategy: dynamic (on demand)
Time stamp counter clock: 1000000 kHz, via system call
System ticker: periodic mode, resolution 10000000 ns
Time partition switch: 10000000 ns, watchdog timeout: 10000000 ns
Time partition synchronization: default
PSP startmode: 0
PSP haltmode: 0
Free memory: 1555588 KiB
PSSW +Ext. FPs +VirtSplit +Messages (Production), Build: D5.0-1674
PIKEOS_MON: Started, version: D5.0-56
Trace Server: version: D5.0-209
guest1(22): Manager starting (partition hwvirt_part:4, process guest1:22)
guest1(22): MANAGER: configuration
guest1(22): VMM init
guest1(22): MANAGER: 2 core(s) detected
guest1(22): Memory init
guest1(22): Threads init
<DRV INFO> eth0: PHY found: NS DP83865 (id=20005c7a)
<DRV INFO> eth0: cpsw: MAC c4:ed:ba:b6:1c:1a assigned to port 0
<DRV INFO> eth0: Registered MAC address(02:70:34:b6:1c:1a) for channel '0'
<DRV INFO> eth0: Registered MAC address(06:70:34:b6:1c:1a) for channel '1'
<DRV INFO> eth0: PHY found: NS DP83865 (id=20005c7a)
<DRV INFO> eth0: cpsw: MAC c4:ed:ba:b6:1c:1b assigned to port 1
<DRV INFO> eth0: Registered MAC address(02:70:34:b6:1c:1b) for channel '2'
<DRV INFO> eth0: Registered MAC address(06:70:34:b6:1c:1b) for channel '3'
CPSW: Provider "eth0" started, Build: D5.0-114 Production
MUXA: Version: D5.0-334
```

```
<DRV INFO> eth0: cpsw: MAC 02:70:34:b6:1c:1a assigned to port 0
<DRV INFO> eth0: cpsw: MAC 06:70:34:b6:1c:1a assigned to port 0
guest1(22): Guest creation
guest1(22): Guest init
8250: Provider "ser0" started, Build: D5.0-198 Production
guest1(22):     guest memory: 0x10000000 in 1 sections
guest1(22): File init
guest1(22): Registers init
guest1(22): Guest core threads creation
guest1(22): Direct-IO init
guest1(22):     IOMMU enabled 0
guest1(22): DTB init
guest1(22): VMM drivers init
guest1(22):     VMM p4bus init
guest1(22):         2 p4bus devices
guest1(22):         64 p4bus operation threads
guest1(22): virtio init
guest1(22):     no areas, virtio disabled.
guest1(22): DTB patching
guest1(22): Manager Memory:
guest1(22):     free 0x0076a000
guest1(22): Hypervisor Memory:
guest1(22):     free 0x0000000000070000
guest1(22): Guest starting.
```

Here is what you will see on the muxa channel:

```
Booting Linux on physical CPU 0x0
Linux version 4.9.120-ELinOS-1734-rt93 (user@host) (gcc version 6.3.0 20170516 (GCC) ) #3 SMP Thu May
CPU: ARMv7 Processor [412fc0f2] revision 2 (ARMv7), cr=10c5387d
CPU: div instructions available: patching division code
CPU: PIPT / VIPT nonaliasing data cache, PIPT instruction cache
OF: fdt:Machine model: PikeOS ARM HW virtualized board
Memory policy: Data cache writealloc
psci: probing for conduit method from DT.
psci: PSCIv1.0 detected in firmware.
psci: Using standard PSCI v0.2 function IDs
psci: MIGRATE_INFO_TYPE not supported.
psci: SMC Calling Convention v1.0
percpu: Embedded 10 pages/cpu @cfde2000 s19852 r0 d21108 u40960
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 65024
Kernel command line: console=vmtty0

PID hash table entries: 1024 (order: 0, 4096 bytes)
Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)
Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory: 249208K/262144K available (3072K kernel code, 75K rwdata, 224K rodata, 6144K init, 218K bss,
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
    vmalloc : 0xd0800000 - 0xff800000   ( 752 MB)
    lowmem  : 0xc0000000 - 0xd0000000   ( 256 MB)
    pkmap   : 0xbfe00000 - 0xc0000000   (   2 MB)
      .text : 0xc0008000 - 0xc0400000   (4064 kB)
```

```
      .init : 0xc0500000 - 0xc0b00000   (6144 kB)
      .data : 0xc0b00000 - 0xc0b12f00   (  76 kB)
       .bss : 0xc0b14000 - 0xc0b4a920   ( 219 kB)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=2, Nodes=1
Hierarchical RCU implementation.
    Build-time adjustment of leaf fanout to 32.
    RCU restricting CPUs from NR_CPUS=8 to nr_cpu_ids=2.
RCU: Adjusting geometry for rcu_fanout_leaf=32, nr_cpu_ids=2
NR_IRQS:16 nr_irqs:16 16
arm_arch_timer: Architected cp15 timer(s) running at 6.14MHz (virt).
clocksource: arch_sys_counter: mask: 0xffffffffffffff max_cycles: 0x16ac02862, max_idle_ns: 440795202
sched_clock: 56 bits at 6MHz, resolution 162ns, wraps every 4398046511085ns
Switching to timer-based delay loop, resolution 162ns
Calibrating delay loop (skipped), value calculated using timer frequency.. 12.28 BogoMIPS (lpj=61440)
pid_max: default: 4096 minimum: 301
Mount-cache hash table entries: 1024 (order: 0, 4096 bytes)
Mountpoint-cache hash table entries: 1024 (order: 0, 4096 bytes)
CPU: Testing write buffer coherency: ok
/cpus/cpu@0 missing clock-frequency property
/cpus/cpu@1 missing clock-frequency property
CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
Setting up static identity map for 0x90100000 - 0x90100058
VMM:  PikeOS HWVIRT Detected, vmm api version 0x13, PikeOS version 0x50003
CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
Brought up 2 CPUs
SMP: Total of 2 processors activated (24.57 BogoMIPS).
CPU: All CPU(s) started in SVC mode.
devtmpfs: initialized
VFP support v0.3: implementor 41 architecture 4 part 30 variant f rev 0
clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 19112604462750000 ns
futex hash table entries: 16 (order: -2, 1024 bytes)
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent allocations
clocksource: Switched to clocksource arch_sys_counter
NET: Registered protocol family 2
TCP established hash table entries: 2048 (order: 1, 8192 bytes)
TCP bind hash table entries: 2048 (order: 2, 16384 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
UDP hash table entries: 128 (order: 0, 4096 bytes)
UDP-Lite hash table entries: 128 (order: 0, 4096 bytes)
NET: Registered protocol family 1
workingset: timestamp_bits=30 max_order=16 bucket_order=0
Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
io scheduler noop registered
io scheduler deadline registered (default)
VMM: probe vmm.
EARLYCON: probe vmm:earlycon.
EARLYCON: registered
VMAPI: probe vmm:vmapi.
P4BUS: init.
P4BUS: init. 10
P4BUS: init. 10
P4BUS: probe vmm:p4bus.
P4BUS_vmchar: init.
```

```
P4BUS_vmtty: init.
P4BUS_vmtty: probe 0.vmtty0.
P4BUS_vmtty: VMTTY 'vmtty0' alias is vmtty0
console [vmtty0] enabled
P4BUS_vmnet: probe 1.eth0.
P4BUS_vmnet: eth0 is registered as eth0 (irq 21)
Registering SWP/SWPB emulation handler
Freeing unused kernel memory: 6144K
can't run '/sbin/netconfig': Permission denied
starting version 232
random: systemd-udevd: uninitialized urandom read (16 bytes read)
random: systemd-udevd: uninitialized urandom read (16 bytes read)
random: udevadm: uninitialized urandom read (16 bytes read)
udev: started


BusyBox v1.22.1 (2019-03-07 21:24:45 CET) built-in shell (ash)
Enter 'help' for a list of built-in commands.

Sleeping 20 seconds to connect to muxa
<DRV INFO> eth0: Link down on slave port 1
<DRV INFO> eth0: Link down on slave port 0
<DRV INFO> eth0: Link up on port 0, 1000 MBPS, full duplex
<DRV INFO> eth0: Link up on port 1, 1000 MBPS, full duplex

================================================================================
==                                                                            ==
==  Welcome to the BusyBox Demo Project!                                      ==
==                                                                            ==
==  The busybox multi-call binary offers you a small unix environment         ==
==  while maintaining a minimum ROM/RAM footprint!                            ==
==                                                                            ==
==  The busybox provides nearly all important unix shell commands.            ==
==  If you want to know which commands are supported by the busybox           ==
==  use the 'busybox' command!                                                ==
==                                                                            ==
================================================================================

/ #
```

11 Board Specific Support

11.1 ARMv7 32Bit Boards

11.1.1 Jetson TK1

The board named “NVIDIA Jetson TK1 developer kit.” (tegra-jetson-tk1-hwvirt) supports PikeOS Virtualization.

11.1.1.1 Supported Boards

The PSP and PikeOS Virtualization have only been tested on the Jetson TK1 board from Nvidia. Any Tegra K1 based board having hardware virtualization support should be supported.

The Tegra Shield tablet is locked in hardware and as a consequence hardware virtualization cannot be used on it.

11.1.1.2 Direct IO Entries

Element Type	Component
Path in pool	board/tegra-k1-hwvirt/tegra-k1-directio.cmp
Name	Tegra K1 Devices

Pool Element 12: Tegra K1 Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

Your Virtualization Process also needs to have the “Direct IO” option activated.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Serial interfaces
- iRam
- Peripheral Bus
- Graphic controller
- Power Management
- AMBA
- PCIE

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.1.1.3 ELinOS Guest

Only the generic “pikeos-hwvirt-secure-v7hf” BSP can be used on the Jetson TK1 board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.1.1.4 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the Jetson TK1 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” enabled PikeOS guest please contact SYSGO Sales.

11.1.2 LS1021a IOT

The board named “NXP LS1021A IOT board.” (ls1021a-iot-hwvirt) supports PikeOS Virtualization.

11.1.2.1 Direct IO Entries

Element Type	Component
Path in pool	board/ls1021a-iot-hwvirt/ls1021a-iot-directio.cmp
Name	ls1021a-iot directio Devices

Pool Element 13: ls1021a-iot directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Base devices
  - Serial
  - Timers
  - SYSCTL
  - DMA
- Additional Devices
  - Ethernet
  - CAN
  - I2C
  - PCI
  - GPIO
  - SATA
  - USB
  - AUDIO
  - VIDEO
  - CRYPT
  - SDHC
  - QSPI

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.1.2.2 ELinOS Guest

Only the generic “pikeos-hwvirt-secure-v7hf” BSP can be used on the LS1021a IOT board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.1.2.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the LS1021a IOT board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.1.3 LS1021a TWR

The board named “NXP LS1021A TWR board.” (ls1021a-twr-hwvirt) supports PikeOS Virtualization.

11.1.3.1 Direct IO Entries

Element Type	Component
Path in pool	board/ls1021a-twr-hwvirt/ls1021a-twr-directio.cmp
Name	ls1021a-twr directio Devices

Pool Element 14: ls1021a-twr directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Base devices
  - Serial
  - Timers
  - SYSCTL
  - DMA
- Additional Devices
  - Ethernet
  - CAN
  - I2C
  - PCI
  - GPIO
  - SATA
  - USB
  - AUDIO
  - VIDEO
  - CRYPT
  - SDHC
  - QSPI

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.1.3.2 ELinOS Guest

Only the generic “pikeos-hwvirt-secure-v7hf” BSP can be used on the LS1021a TWR board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.1.3.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the LS1021a TWR board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.1.4 Renesas R-Car H2

The board named “Renesas R-Car H2 based on ARM Cortex A15 Board.” (renesas-rcarh2-hwvirt) supports PikeOS Virtualization.

11.1.4.1 Direct IO Entries

Element Type	Component
Path in pool	board/renesas-rcarh2-hwvirt/renesas-rcarh2-directio.cmp
Name	Renesas-rcarh2 Devices

Pool Element 15: Renesas-rcarh2 Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Serial
- USB
- Video
- GPIO
- MMC
- SATA
- SD card
- i2c
- DMA
- Others (available in the main tab of the component)

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.1.4.2 ELinOS Guest

Only the generic “pikeos-hwvirt-secure-v7hf” BSP can be used on the Renesas R-Car H2 board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.1.4.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the Renesas R-Car H2 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.1.5 VAYU UEVM

The board named “VAYU Evaluation Module.” (vayu_uevm-hwvirt) supports PikeOS Virtualization.

11.1.5.1 Supported Boards

This board is also named Jacinto 6.

11.1.5.2 Direct IO Entries

Element Type	Component
Path in pool	board/omap543x-hwvirt/vayu_uevm-directio.cmp
Name	Vayu_uevm directio Devices

Pool Element 16: Vayu_uevm directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Smartflex
- Timers
- MMC
- GPIO
- I2C
- MailBox
- Video Input Capture
- Multichannel Audio Serial port
- Multichannel Serial peripheral interface
- Serial interface
- Ethernet
- USB
- Others (available in the main tab of the component)

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.1.5.3 ELinOS Guest

Only the generic “pikeos-hwvirt-secure-v7hf” BSP can be used on the VAYU UEVM board. It can only use P4Bus for communication.

Note: On ELinOS 6.0, a BSP based on the 3.8 Linux kernel is supported.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.1.5.4 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the VAYU UEVM board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.1.6 VPX3-1701

The board named “VPX3-1701 3U VPX ARM Cortex A7 SBC.” (vpx3-1701-hwvirt) supports PikeOS Virtualization.

11.1.6.1 Direct IO Entries

The PSP does not provide a component defining the existing IO. You can use the “hwvirt custom directIO” component as described in section 3.10 to define them if needed.

11.1.6.2 ELinOS Guest

Only the generic “pikeos-hwvirt-secure-v7hf” BSP can be used on the VPX3-1701 board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.1.6.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the VPX3-1701 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.1.7 TI keystone2

The board named “Texas Instrument Keystone2 K2HK EVM.” (ti-keystone2-hwvirt) supports PikeOS Virtualization.

11.1.7.1 Direct IO Entries

Element Type	Component
Path in pool	board/ti-keystone2-hwvirt/ti-keystone2-directio.cmp
Name	ti-keystone2 directio Devices

Pool Element 17: ti-keystone2 directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Ethernet
- IPC irq
- Clock
- PCI
- PDSP
- AEMIF
- Serial
- GPIO
- USB
- Hardware crypto engine
- NAND
- Timer-64
- Packet DMA
- System Trace manager

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.1.7.2 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the TI keystone2 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.2 ARMv8 64Bit Boards

11.2.1 Juno A57/A53

The board named “ARM Juno Board with Cortex A57/A53.” (juno-a57-hwvirt) supports PikeOS Virtualization.

11.2.1.1 Direct IO Entries

Element Type	Component
Path in pool	board/juno-a57-hwvirt/juno-a57-directio.cmp
Name	Juno a57 directio Devices

Pool Element 18: Juno a57 directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Ethernet
- Serial
- Timers
- USB
- I2C
- Others (available in the main tab of the component)

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.2.1.2 ELinOS Guest

The BSP “juno-a57_hwvirt” can be used as PikeOS guest on this board. It supports Ethernet, Serial and USB as well as P4Bus communication.

The generic “pikeos-hwvirt-secure-v8hf” BSP can also be used on the Juno A57/A53 board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.2.1.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the Juno A57/A53 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.2.2 Foundation Platform

The board named “ARM Foundation platform simulator (ARMv8) with Hardware Virtualization.” (foundation-armv8-hwvirt) supports PikeOS Virtualization.

11.2.2.1 Direct IO Entries

Element Type	Component
Path in pool	board/foundation-armv8-hwvirt/foundation-armv8-directio.cmp
Name	foundation platform armv8 directio Devices

Pool Element 19: foundation platform armv8 directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

Your Virtualization Process also needs to have the “Direct IO” option activated.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Ethernet
- Serial
- Others (available in the main tab of the component)

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.2.2.2 ELinOS Guest

The BSP “foundation-v8_hwvirt” BSP can be used on as PikeOS guest on this board. It supports Ethernet and Serial as well as P4Bus communication.

The generic “pikeos-hwvirt-secure-v8hf” BSP can also be used on the Foundation Platform board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.2.2.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the Foundation Platform board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” enabled PikeOS guest please contact SYSGO Sales.

11.2.3 FVP A57/A53

The board named “Fixed Virtual Platform (FVP) using Cortex A5x Core(s).” (fvp-a5x) supports PikeOS Virtualization.

11.2.3.1 Supported Boards

The BSP has been tested using Fixed Virtual Platform with the A57/A53 cores (up to 4 A57 with 4 A53). It should also work using a model in Fastmodel using combinations of those types of cores.

11.2.3.2 Direct IO Entries

Element Type	Component
Path in pool	board/fastmodel-a5x-hwvirt/fastmodel-a5x-directio.cmp
Name	FVP for Cortex A5x Direct-I/O Devices

Pool Element 20: FVP for Cortex A5x Direct-I/O Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Ethernet
- Serial
- Others (available in the main tab of the component)

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.2.3.3 ELinOS Guest

The BSP “fvp-a57_hwvirt” BSP can be used on as PikeOS guest on this board. It supports Ethernet and Serial as well as P4Bus communication.

The generic “pikeos-hwvirt-secure-v8hf” BSP can also be used on the FVP A57/A53 board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.2.3.4 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the FVP A57/A53 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.2.4 LS1043A

The board named “QorIQ LS1043A.” (ls1043a-rdb-hwvirt) supports PikeOS Virtualization.

11.2.4.1 Direct IO Entries

Element Type	Component
Path in pool	board/ls1043a-rdb-hwvirt/ls1043a-rdb-directio.cmp
Name	ls1043a-rdb directio Devices

Pool Element 21: ls1043a-rdb directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines one global option:

- Enable all devices: Enables all devices listed in the component.

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.2.4.2 ELinOS Guest

For this BSP, there is **NO ELinOS support**. The Linux kernel used comes directly from NXP/Freescale.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.2.4.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the LS1043A board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.2.5 Zynq ZCU102

The board named “XilinX Zynq ZCU102.” (zynq-zcu102-hwvirt) supports PikeOS Virtualization.

11.2.5.1 Direct IO Entries

Element Type	Component
Path in pool	board/lzynq-zcu102-hwvirt/zynq-zcu102-directio.cmp
Name	Zynq ZCU102 directio Devices

Pool Element 22: Zynq ZCU102 directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Ethernet
- Serial
- Timers
- USB
- I2C
- CAN
- QSPI
- PCIE
- Others (available in the main tab of the component)

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.2.5.2 VGIC specific configuration for Linux guest

For this board, the GIC address contained in the DTB is not the one used by the GIC driver. That means, **only for a DirectIO guest**, the VGIC must not be mapped at the “real physical address” contained in the DTB but at the “real physical address”. To achieve this the following modification is required:

- Disable the “Use GIC real physical values” option (see the section 3.5.5).
- Set the “GIC Contoller Dist address” to the one configured in the Zynq ZCU102 PSP (0xf9010000).
- Set the “GIC Contoller CPU address” to the one configured in the Zynq ZCU102 PSP (0xf9020000).

11.2.5.3 ELinOS Guest

The BSP “zynq-zcu102_hwvirt” can be used as PikeOS guest on this board. It supports Ethernet, Serial and USB as well as P4Bus communication. The generic “pikeos-hwvirt-secure-v8hf” BSP can also be used on the Zynq ZCU102 board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.2.5.4 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the Zynq ZCU102 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.2.6 Jeston TX1

The board named “NVIDIA Jetson TX1 developer kit (Tegra X1 Processor).” (tegra-jetson-tx1-hwvirt) supports PikeOS Virtualization.

11.2.6.1 Direct IO Entries

Element Type	Component
Path in pool	board/tegra-jetson-tx1-hwvirt/tegra-jetson-tx1-directio.cmp
Name	Tegra TX1 directio Devices

Pool Element 23: Tegra TX1 directio Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines two global options:

- Enable all devices: Enables all devices listed in the component.
- Enable Linux base devices: Enables only the devices needed by Linux to boot (for a Linux without any devices activated, otherwise the corresponding devices need to be allowed in the configuration).

All supported devices are ordered by subsection:

- Serial
- I2C
- SPI
- Timer
- Watchdog
- GPIO
- RTC
- SDMMC
- USB
- Others (available in the main tab of the component)

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.2.6.2 ELinOS Guest

Only the generic “pikeos-hwvirt-secure-v8hf” BSP can be used on the Jeston TX1 board. It can only use P4Bus for communication.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.2.6.3 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the Jeston TX1 board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

11.2.7 Renesas Salvator X

The board named “Renesas Salvator X (rcarH3.” (renesas-salvator-x-hwvirt) supports PikeOS Virtualization.

11.2.7.1 Direct IO Entries

Element Type	Component
Path in pool	board/renesas-salvator-x-hwvirt/renesas-salvator-x-directio.cmp
Name	Renesas Salvator X Direct IO Devices

Pool Element 24: Renesas Salvator X Direct IO Devices

The PSP defines most of the existing IO on the board in its “directio” component. You will need to add this component to your project and assign it to your guest to allow direct usage of devices by a guest.

This component defines one global option:

- Enable all devices: Enables all devices listed in the component.

Note: The integrator can configure more devices using the “hwvirt custom directIO” component as described in section 3.10

11.2.7.2 VGIC specific configuration for Linux guest

For this board, the GIC address contained in the DTB is not the one used by the GIC driver. Before accessing the “GIC CPU” base registers the GIC driver adds an offset of 0xf000. That means, **only for a DirectIO guest**, the VGIC must not be mapped at the “real physical address” contained in the DTB but at the “real physical address” plus an offset of 0xf000. To achieve this the following modification is required:

- Disable the “Use GIC real physical values” option (see the section 3.5.5).
- Set the “GIC Contoller Dist address” to the one configured in the Renesas Salvator X PSP (0xf1010000).
- Set the “GIC Contoller CPU address” to the one configured in the Renesas Salvator X PSP (0xf102f000).

11.2.7.3 ELinOS Guest

For this BSP, there is **NO ELinOS support**. Only the “pikeos-hwvirt-secure-v8hf” BSP is supported. The Linux kernel (Yocto) from Reneses can be adapted to run over the PikeOS Virtualization.

Note: A standard Linux can also be used as guest. Please refer to section 7.1 and follow the standard procedure for it.

11.2.7.4 PikeOS Guest

Only the generic PikeOS hardware virtualization PSP can be used on the Renesas Salvator X board. It can only use P4Bus for communication.

Note: If you need to have a “Direct IO” PikeOS guest please contact SYSGO Sales.

12 Common Use Cases and Troubleshooting

This chapter describes the most common use cases, the most common problems when using PikeOS Virtualization and how they can be solved or investigated.

12.1 Common Use Cases

12.1.1 Virtual Ethernet Communication between several Linux Guests

This section describes the configuration of a virtual ethernet connection between several guests by using the veth PikeOS driver and p4bus-devices.

- **Linux guest side:**
  - **Create a new "ElinOS System Project":** Create a new "ElinOS System Project" based on a template (BusyBox) and select a PikeOS hardware virtualization SecureIO BSP ("PikeOS secure I/O on ARM v7/v8 32bit (PikeOS HWVIRT)" for example). The boot strategy needs to be set as "pikeos_hwvirt".
  - **ElinOS guest configuration:** In the feature configurator, the p4bus support needs to be activated in:

    `"PikeOS virtualization/P4bus support"`

    Activate the "Basic networking support (TCP/IP)" under the Networking option, this will activate automatically the "Network over P4BUS" option in:

    `"Networking/Network driver options/Ethernet networking support"`

    At this point all drivers required to have a network connection over the p4bus are enabled but we need to configure the IP address for the guest. The IP address configuration can be set in:

    `"Networking/ TCP/IP setting/Choose your setup method"`

    The easiest way is to configure it with fixed values (IP address, hostname, netmask and gateway). Then just add the network tools, clients and server needed (SSH for example).
    Please reproduce the previous steps for the other guests while keeping in mind to change the TCP/IP settings for all of them.
- **Integration side:**
  - **Create a new "PikeOS integration project":** Create an integration project based on the "empty" demo and using a board with the hardware virtualization support.
  - **Add the PikeOS Driver:** Depending on the selected board, you can have the choice to use:
    - **A PikeOS Ethernet driver:** It provides a real ethernet connection to the guests by using the network device present on the board.
    - **A PikeOS Virtual Ethernet driver "veth-network-driver":** It provides a virtual network connection that can be used between the guests.

    In the Overview tab from your project, add a new component from:

    `"PIKEOS_POOL/driver/ethernet"`
  - **Add hardware virtualization guests:** In the Overview tab of your integration project, add a new hardware virtualization guest from:

    `"PIKEOS_POOL/virtualization/HWVIRT linux with DTB"`

Please reproduce this step while taking care to rename the components if the integration project contains more than one guest.

- **Add the p4bus-device for the ethernet connection:** For each guest, add a p4bus-device component from:

  `"PIKEOS_POOL/virtualization/p4bus/p4bus device"`

  The new p4bus-device needs to be attached to the guest's hardware virtualization partition and needs to be configured as follows:

  - **Host Type** vmfile
  - **Use a File Provider** true
  - **Guest Type** vmnet
  - **Guest Name** "name of your device"

  By checking the option "Use a File Provider" the p4bus-device requires a FILE dependency that should be satisfied by a free virtual channel provided by the PikeOS Ethernet driver (" veth-vchan0-channel" for example).

At this point you just have to compile your ELinOS projects, configure the hardware virtualization guests (binaries, DTBs...) and compile your integration project to have a bootable image.

12.1.2 Queuing Port Communication Channel between two Linux Guests using a Char Interface

This section describes the configuration of a queuing port channel between two guests by using a char interface.

- **Linux guest side:**
  - **Create a new "ElinOS System Project":** Create a new "ElinOS System Project" based on a template (BusyBox) and select a PikeOS hardware virtualization SecureIO BSP ("PikeOS secure I/O on ARM v7/v8 32bit (PikeOS HWVIRT)" for example). The boot strategy needs to be set as "pikeos_hwvirt".
  - **ElinOS guest configuration:** In the feature configurator, the p4bus support needs to be activated in:

    `"PikeOS virtualization/P4bus support"`

    Activate the "Char over P4Bus" option in:

    `"PikeOS virtualization/P4bus support/Char over P4Bus"`

    Activate the "Console over P4Bus TTY" option in:

    `Console`

    At this point all drivers required are included in the guest. Please reproduce the previous steps for the other guests.
- **Integration side:**
  - **Create a new "PikeOS integration project":** Create an integration project based on the "empty" demo and using a board with the hardware virtualization support.
  - **Add the PikeOS Ethernet Driver**
  - **Add the MUXA:** Configure the MUXA to use the PikeOS Ethernet Driver from your board.
  - **Add hardware virtualization guests:** In the Overview tab of your integration project, add a new hardware virtualization guest from:

    `"PIKEOS_POOL/virtualization/HWVIRT linux with DTB"`

    Please reproduce this step while taking care to rename the components.
  - **Add a p4bus-device for the queuing port communication[1/2]:** For the first guest, add a p4bus-device component from:

    `"PIKEOS_POOL/virtualization/p4bus/p4bus device"`

The new p4bus-device needs to be attached to the guest's hardware virtualization partition and needs to be configured as following:

- **Host Type** vmqport
- **Ports configuration** connected to an other p4bus device
- **Slave p4bus device** false
- **Max message size** 1000
- **Max message count** 256
- **Guest Type** vmchar
- **Guest Name** my_char

- **Add a p4bus-device for the queuing port communication[2/2]:** For the second guest, add a p4bus-device component from:

  `"PIKEOS_POOL/virtualization/p4bus/p4bus device"`

  The new p4bus-device needs to be attached to the guest's hardware virtualization partition and needs to be configured as follows:

  - **Host Type** vmqport
  - **Ports configuration** connected to an other p4bus device
  - **Slave p4bus device** true
  - **Guest Type** vmchar
  - **Guest Name** my_char

  By setting the option "Ports configuration" to "connected to an other p4bus device" and the option "Slave p4bus device" to true, this device will require a dependency with an other p4bus-device. This dependency needs to be satisfied by the vmqport p4bus-device from the first guest. Once the dependency is satisfied, the two queuing ports and the channel will be created automatically according to the parameters configured in the first guest.

At this point you just have to compile your ELinOS projects, configure the hardware virtualization guests (binaries, DTBs...) and compile your integration project to have a bootable image.

Once done, run the MUXA with two telnet windows connected to the guests consoles. You will find the char interface linked to the queuing ports in:

```
"dev/vmchar0
```

You can test the communication with the following steps:

```
guest1# echo "Hello guest2 I am guest1" > /dev/vmchar0

guest2# cat /dev/vmchar0
Hello guest2 I am guest1
```

12.1.3 Grant Access for a new Device to a Linux DirectIO Guest

This section describes the procedure to follow in order to grant the access of a physical device to a Linux directIO guest. This is a common use case when you have a DirectIO guest and the following message is printed on the console:

```
Linux-DTB(22): guest did a 8 bits write to 0x70006310 with value 0x0 at 0xc0146528
Linux-DTB(22): Exception on core 0 at 0xc0146528: [Unhandled abort exception]
Linux-DTB(22): Guest to access data at invalid address 0x70006310
Linux-DTB(22): pc = 0xc0146528 lr = 0xc0145e64 sp = 0xcf83bb58
Linux-DTB(22): exception type: 0x2
Linux-DTB(22): exception syndrome: 0x93060047
Linux-DTB(22): r00 = 0xc09ef5e4 r01 = 0x00000004 r02 = 0x00000000 r03 = 0xfe006300
```

```
Linux-DTB(22): r04 = 0x00000010 r05 = 0xc09ef5e4 r06 = 0x00000000 r07 = 0xcf9dac9c
Linux-DTB(22): r08 = 0x40000053 r09 = 0xcf92e1c0 r10 = 0xc026884c r11 = 0x01851960
Linux-DTB(22): r12 = 0x00000000
```

This message means that the guest OS tries to access to an address (0x70006310 here) which is not granted for it. Firstly, you need to check if the “Devices” component related to the board is included in the integration project and if the accesses for all devices are granted. If not, then the guest OS tries to access a non standard device. This case can be handled by different ways depending of your needs:

- **You want to give the access to the device:** In this case the integrator needs to add a “hwvirt custom direct IO” component from:

  `"PIKEOS_POOL/virtualization/hwvirt/hwvirt custom direct IO"`

  This component represents a device (address, size and IRQ) for which the access will be granted to the guest.

- **You want to simulate the device:** In this case the integrator needs to add a “Virtualization virtual IO” from:

  `"PIKEOS_POOL/virtualization/misc/Virtualization virtual IO"`

  This component simulates a device either with a memory area accessible or by a null device.

Both components need to be attached to the guest’s hardware virtualization partition.

12.1.4 Use Virtualization drivers version provided with PikeOS instead of ELinOS

This section describes the procedure to follow in order to use the PikeOS virtualization drivers in a Linux guest instead of default drivers included in ELinOS. This can be needed in case your PikeOS version is newer then your ELinOS version (usually due to a PikeOS service release or some custom drivers provided in a project).

- **Import virtualization drivers in ELinOS project with CODEO:** Import drivers in “kernelsrc” project folder:
- Create a “drivers” folder in “kernelsrc” folder
  - Right click on “kernelsrc” folder
  - New
  - Folder
  - Folder name “drivers”
- Create a “virt” folder in “drivers” folder
  - Right click on “drivers” folder
  - New
  - Folder
  - Folder name “virt”
- Import “drivers” folder in “virt” folder
  - Right click on “virt” folder
  - Import
  - Select General -> File System -> Next
  - Browse -> opt -> pikeos-X.X -> share -> hwvirt-linux -> pikeos

- **Import virtualization drivers in ELinOS project with Command Line:** Import drivers in “kernelsrc” project folder:
  - cd /workspace/PROJECT_NAME.app/kernelsrc
  - mkdir drivers
  - cd drivers
  - mkdir virt
  - cd virt
  - cp -r opt/pikeos-X.X/share/hwvirt-linux/pikeos .

This drivers version of PikeOS will be use instead of default version found in ELinOS.

12.2 Common Debug Good Practices

This section describes the good practices to obtain useful information when a guest is crashing.

- **Set the guest in verbose or debug mode:** This practice allows you to get useful information about the guest configuration as:
  - p4bus devices registered and configured for the guest
  - DirectIO devices granted to the guest
  - Guest and Manager memory configuration and mapping
  - Start addresses for all cores
  - Context dump when an exception occurs

  This can be set by the option “Log Level” in the hardware virtualization process component.
- **Set the print frequency to get the PC and context:** This option will print at the specified frequency the current address of the core and if the “Log Level” option is set to “debug” will also print the content of the core registers. This practice is useful when the guest is stuck and nothing is printed on its console (WFI for example). This can be set by the option “Debug Print Frequency (ms)” in the subcategory “Debugging” from the hardware virtualization process component.
- **Hypervisor debug call:** The Hypervisor debug call is a specific HVC call caught by the hypervisor which will print the current context of the core. This will print the same output as the “Debug Print Frequency (ms)” option but the call can be done everywhere from the code. The following inline function must be called in your application including the “vmm.h” header.

  ```
  static inline void vmm_debug(unsigned long arg0, unsigned long arg1, unsigned long arg2);
  ```

  The function arguments “arg0”, “arg1” and “arg2” can be set with the value you want and will be respectively pushed in “r0/x0”, “r1/x1” and “r2/x2” registers.

  Once called, the HVC exception will be caught by the hardware virtualization manager which will print the current core context as the following:

  ```
  guest1(42): Exception on core 1 at 0xc01262c8: [Debug HVC Call]
  guest1(42): pc = 0xc01262c8 lr = 0xc00a0924 sp = 0xc6863e00
  guest1(42): exception type: 0x1
  guest1(42): exception syndrome: 0x4a000063
  guest1(42): r00 = 0x00000000 r01 = 0x00000000 r02 = 0x00000000 r03 = 0xc01262b8
  guest1(42): r04 = 0xc0561f88 r05 = 0x00000000 r06 = 0xc6ff6e00 r07 = 0x00000000
  guest1(42): r08 = 0xc740aea0 r09 = 0xc6864cc0 r10 = 0xc740b000 r11 = 0x00000000
  guest1(42): r12 = 0x00050005
  ```

12.3 Common Error Messages

Hereafter a list of common error messages that are raised because of problem of configuration. For a more exhaustive list of errors and there messages see section 14.

- **Error: cannot create guest**

  This error occurs at start-up of a guest partition. It means the PSP you are using does not have hardware virtualization support.

  Check your integration project's PSP and select a board with the "with HWVIRT" suffix.

- **Error: Error mapping memory to guest**

  It is not possible to map memory into the guest. The hypervisor probably does not have enough memory to create the required page tables.

  You can try to increase the **hypervisor memory size** parameter in the hardware virtualization process options (Advanced parameters options).

- **Error: Error mapping directio entry XX io**

  It is not possible to map IO memory into the guest. The hypervisor probably does not have enough memory to create the required page tables.

  You can try to increase the **hypervisor memory size** parameter in the hardware virtualization process options (Advanced parameters options).

- **Error: Error forwarding IRQ of DirectIO entry XX**

  It is not possible to forward a hardware interrupt to your guest. There are probably two virtual machines on your system that want to have the same interrupt number forwarded, which is not possible. It can also be that an other PikeOS driver is using the given interrupt.

- **Error: "MANAGER: guest tried to overmmap the VGIC registers"**

  This error is raised by the manager when checking all the mappings for the guest.

  It means that the guest request a mapping (or a mapping is configured in the integration project) which

  will overmap the "Virtual GIC controller" of the guest. This mapping is forbidden.

  Note: To get more informations about the addresses of the error, configure the guest verbosity to **"Verbose"**.

- **Error: "Cannot map over GIC addresses"**

  This error is raised by the manager when checking all the mappings for the guest.

  It means that the guest request a mapping (or a mapping is configured in the integration project) to mmap

  the "Interrupt controller" in the guest. The real "interrupt controller" is reserved for PikeOS and

  does not have to be mapped in the guest.

  Note: To get more informations about the addresses of the error, configure the guest verbosity to **"Verbose"**.

- **Error: Guest is back on core X at 0x...**

  The guest has been stopped with an unsupported exception.

  Most common case here is when a guest is trying to execute code at an invalid address.

- **Error: Invalid dtb header**

  The DTB file that you gave has not been recognized by the manager. Make sure you provided a DTB file and not a DTS.

  You can validate that the DTB file is correct using the "dtc" command on your development host:

```
dtc -I dtb -O dts -o test.dts GUESTNAME.dtb
```

- **Error: The instruction decoder does not support the instruction which generated the abort**

  This error is raised when an abort occurs and the instruction is not supported by the decoder. The Instruction decoder cannot resolve instructions which are dealing with more than one register such as:

  - pop
  - push
  - stm
  - ldm

- **Error: Error no operation threads left to execute the XXXX operation**

  This error is raised when an application has executed a blocking operation on a P4 Bus device and there is no operation thread available in the manager to execute it. To avoid this kind of error, increase the number of P4 Bus operation threads in your integration project. Please refer to the section 3.5.

- **Error: vmtty 0.vmtty0: P4BUS: No interrupt resource**

  This error can only be seen in a Linux guest and indicates that there is a mistake in the interrupt controller declaration in the DTB. It's mainly due to the GIC node synthax in the DTB, please refer to the section 7.1.4.2.

  Note: This example only shows an error on the p4bus-vmtty driver but could be raised by other drivers. Please only take into account the **No interrupt resource**.

- **Error: Accessing the GIC address for a DirectIO Guest**

  The following error message can occur booting a DirectIO guest (i.e. the Linux kernel provided by the board manufacturer):

```
Linux-DTB(22): guest did a 32 bits read from 0x2c010004 at 0xffffffc00033c758
Linux-DTB(22): Exception on core 0 at 0xffffffc00033c758: [Unhandled abort exception]
Linux-DTB(22): Guest to access data at invalid address 0x2c010004
Linux-DTB(22): pc = 0xffffffc00033c758 lr = 0xffffffc00033c960 sp = 0xffffffc00084be60
Linux-DTB(22): exception type: 0x2
Linux-DTB(22): exception syndrome: 0x93810006
Linux-DTB(22): x00 = 0xffffffc00fff5098 x01 = 0xffffff8000000004 x02 = 0xffffff8000000000
Linux-DTB(22): x03 = 0x00000000ffffffff x04 = 0x0000000000000000 x05 = 0xffffffc00fff5098
Linux-DTB(22): x06 = 0x0d111848efe6e6f3 x07 = 0xfefefeff646c606d x08 = 0x7f7f7f7f7f7f7f7f
Linux-DTB(22): x09 = 0xfffffffffffffffb x10 = 0x0101010101010101 x11 = 0x0000000000000004
Linux-DTB(22): x12 = 0x0000000000000010 x13 = 0x0000000000000000 x14 = 0xffffffc000866b00
Linux-DTB(22): x15 = 0x0000000000000000 x16 = 0x0000000000000014 x17 = 0x0000000000000c7f
Linux-DTB(22): x18 = 0x0000000000000000 x19 = 0xffffffc00084eb98 x20 = 0xffffffc00084eba0
Linux-DTB(22): x21 = 0xffffffc00084eba0 x22 = 0x0000000000000000 x23 = 0xffffffc00084e000
Linux-DTB(22): x24 = 0x00000000ffffffff x25 = 0x0000000000000000 x26 = 0x0000000000000000
Linux-DTB(22): x27 = 0xffffffc000081220 x28 = 0x0000000000000000 x29 = 0xffffffc00084be60
Linux-DTB(22): Stopping guest.
```

  This error is raised when the guest tries to access an address which is not mapped. It is usually printed when the guest OS accesses a peripheral which is not configured as a DirectIO device.

  If the address causing the error is the GIC address (here "0x2c010004" for the Juno-a57 board), that means that the guest configuration is wrong and the option **"Use GIC real physical values"** must be set to true. Please refer to the section 3.5.5.

12.4 Common Information Messages

- **Message: Invalid CP15 access**

  The guest tried to access the given CP15 register but it does not have the right to do so. Access has been emulated.

  You can turn this message off by decreasing the log level to “info”.

- **Message: Forbidden SMC instruction**

  The guest wants to use a Trustzone system call which is not possible as virtualized guest. Request has been ignored and guest is continuing execution as if nothing happened.

  You can turn this message off by decreasing the log level to “info”.

- **Message: [Breakpoint] Exception on core X at 0x.....**

  You have activated the debug print frequency and this is the current execution address of your guest.

  To turn this off set the print frequency parameter back to 0.

12.5 Common Problems

- **I only have a message saying “Manager starting” and nothing else**

  Your guest can be running or not, it can have simply no console or be looping waiting for something.

  To investigate, the following should be tried or checked:

  - Turn guest log level to verbose or debug.
  - Check that your guest has a console and has access to it.
  - Turn on debug frequency and check the execution addresses.
  - Put some debug code in your guest boot sequence.

- **System completely blocked with 2 SMP guests**

  When you have two or more guests, and each guest is using more than one core, you can end up in a deadlock if the two guests have the same priority.

  This happens when the following scenario occurs (or something equivalent):

  - Guest 1 core 0 took a lock and has been preempted by PikeOS to execute guest 2 core 0.
  - Guest 2 core 1 took a lock and has been preempted by PikeOS to execute guest 1 core 0
  - Guest 1 core 1 is waiting for the lock taken by core 0 in a busy loop.
  - Guest 2 core 0 is waiting for the lock taken by core 1 in a busy loop.

  In this scenario, on core 0, guest 2 core 0 is executing while on core 1, guest 1 core 1 is executing.

  As the PikeOS scheduler is a FIFO scheduler, the cores in the loop waiting for a lock to be released will not allow anyone at the same priority to be scheduled until the lock is released. As all cores are at the same priority in this scenario, the lock will have no chance to be released which is a dead lock condition.

  To solve this you must make sure your two guests do not have the same priority or are in different time partitions. This can be done by changing one of the guest processes’ “Priority” parameters in the hardware virtualization process options (Advanced parameters options).

- **P4 PANIC using a Direct IO Guest**

  A P4 PANIC can be raised using a Direct IO guest. This can be due to a configuration error if a guest driver (for a device configured with Direct IO) has some accesses to the clocks and changes them.

  All clock accesses done by the guest drivers must be known by the integrator to identify which driver can be problematic for the whole system.

- **The guest does not boot, the last message is “Start core 0 at 0x90008000”**

  If the guest kernel image is too large (for example, including the rootfs), the decompression of the kernel could overwrite the DTB loaded in RAM.

  To avoid this, the image size must be reduced by:

  - Remove the root file system from the kernel image.
  - Compile some drivers as modules.

13 Known Bugs and Limitations

This chapter lists the known bugs and limitations of this version of PikeOS Virtualization support.

- **Direct IO interrupt response time**

  When using native drivers in a guest, the performances can be decreased, in comparison to the native performances of the driver, for devices generating a great number of interrupts. The interrupt forwarding overhead is currently high.

- **GIC Version 3 not supported on guests**

  PikeOS Hardware Virtualization emulates only a GIC version 2 to guests (host side of PikeOS properly supports both GIC version 2 and 3). As a consequence, you must make sure that your guest properly supports a version 2 of the GIC controller. Linux does support both versions and your DTB must have a GIC node compatible with a GIC version 2 (the *compatible* field must not be set to *gic-v3*).

14 User and Kernel Level API Description

This section describes the API provided by the PikeOS hardware virtualization.

14.1 HWVIRT Errors

This section describes all error codes in the hypervisor and the manager and gives some extensive help on the possible reasons for each error to happens.

The hypervisor errors might be returned to external exception handlers.

14.1.1 Defines

P4HWVIRT_E_GROUP_MASK

P4HWVIRT Error group mask.

Description:

This is made to be compatible with IPC error types so that PikeOS kernel is letting those error go through the kdev interface.

P4HWVIRT_E_MASK

P4HWVIRT Error mask.

Description:

This is compatible with standard PikeOS error numbers/

14.1.2 Enumerations

Enumeration type *p4hwvirt_e_hyp_t*

Hypervisor errors.

This is listing all errors that can be returned by the hypervisor.

If an error not listed here is present, this means a generic PikeOS error code has been returned from vm_open or vm_ioctl and the standard PikeOS reference manual must be checked to have more information.

All errors returned by the hypervisor are startint at 0x10000.

Name	Description
P4HWVIRT_E_OK	
P4HWVIRT_E_HYP_GROUP	0x10000: Hypervisor Group Number. This error code is never returned but can be used in health monitoring to define a common behaviour for all errors generated by the hypervisor.
P4HWVIRT_E_HYP_INIT_HYPERVISOR	0x10001: Error during low level hypervisor init. This error happens when the hypervisor mode is initialized on a core. This can be due to the fact that PikeOS was not started in hypervisor mode. This error is only raised internally during the kdev init.
P4HWVIRT_E_HYP_INIT_INVALID_MEMORY	0x10002: Error during hypervisor memory init. This error can be raised when the hypervisor mode page tables are created if it is not possible to flush the memory of the page tables. This error is only raised internally during the kdev init.
P4HWVIRT_E_HYP_INIT_INVALID_CONFIG	0x10003: Error during hypervisor init when scanning the configuration in the property file system. This happens if some entries are missing or have unsupported values in the property file system. This error is only raised internally during the kdev init.
P4HWVIRT_E_HYP_INIT_TIMER_NOFREQ	0x10004: Error during hypervisor init if the timer frequency cannot be auto-detected. This error is only raised internally during the kdev init.
P4HWVIRT_E_HYP_OPEN_INVALID_GUEST_NAME	0x10005: Invalid Guest name. The name of the guest which was opened is not valid. Several reasons are possible here: • the name is to long (more then 32 characters) • the name is empty • there is no such guest configured in the hwvirt provider. In this case you can check if the property file system has a directory provider/<providername>/<guestname> containing an entry "partid" and an entry "hypmem".

P4HWVIRT_E_HYP_OPEN_BAD_PARTITION	0x10006: Invalid partition ID for the guest. The partition ID of the client which opened the guest is different from the partition ID configured for the guest in the property file system.
P4HWVIRT_E_HYP_OPEN_BAD_TASK	0x10007: Invalid task ID for the guest. The task ID of the client which opened the guest is different from the task ID of the client that previously opened this guest. Once a guest has been assigned to a task ID it cannot be used by an other task even in the same partition.
P4HWVIRT_E_HYP_OPEN_ALREADY_OPENED	0x10008: Invalid reopen of an already opened guest. The guest that is being opened using a vm_open is already opened in the same task. This means that the guest was not properly closed before reopening it or that a second opened is being done on the same guest. Only one open can be done at a time on a guest. The file descriptor should be reused internally in a task instead of reopening it.
P4HWVIRT_E_HYP_OPEN_NULL_GUEST	0x10009: NULL guest detected during open. The VMIT configuration contains a gate with a non NULL name entry which makes the kdev receive an open without a name. This is an unsupported configuration mode for the hypervisor. Only gate without names are supported by the hypervisor.
P4HWVIRT_E_HYP_INVALID_GUEST_STATE	0x1000a: Invalid guest state. The guest is not currently in a state allowing this operation. This can happen when trying to request operations on a guest that was not properly initialized or opened by the manager. This can happen if: • a guest init is done twice • an operation is done on a closed or non initialised guest
P4HWVIRT_E_HYP_INVALID_GUEST_COREID	0x1000b: Invalid core ID. An operation was requested on a core number which is not valid for the guest. This is for example generating an interrupt on a guest core ID higher then the total number of cores of the guest.
P4HWVIRT_E_HYP_NO_HYPMEM	0x1000c: Not enough hypervisor memory. There is not enough hypervisor memory for this operation. This is usually the case during init if there is not enough space to allocated internal structures or to allocate page table for the guest mappings. This can also happen at runtime if the guest is trying to map some extra area (during an mmap call usually). This can be solved by increasing the amount of hypervisor memory allocated to this guest in the configuration.

P4HWVIRT_E_HYP_IOCTL_UNSUPP	0x1000d: Unsupported hypervisor IOCTL. The IOCTL number done by the user task is not supported by the hypervisor. This probably means that your version of the manager is not compatible with this version of the hypervisor.
P4HWVIRT_E_HYP_IOCTL_INVALID_ARGUMENT	0x1000e: Invalid IOCTL argument type or size. The argument provided to an IOCTL is not of the required type or size.
P4HWVIRT_E_HYP_IOCTL_INVALID_USER_ADDRESS	0x1000f: Invalid user address in IOCTL argument. The argument provided to an IOCTL contained an address invalid in the calling user application.
P4HWVIRT_E_HYP_MAP_INVALID_ADDRESS	0x10010: Cannot map to a guest an inaccessible user address. When requesting to map an area of memory or IO to a guest, the given area (address and size) must be mapped and accessible to the calling application. Here the passed area was not completely accessible to the calling user application.
P4HWVIRT_E_HYP_MAP_OVERMAP	0x10011: Error during mapping due to an overmap. An error was raised during a mapping operation because a there is already a mapping at the given address and this mapping is not compatible with the wanted one. This can be due to a different physical address or due to different mapping attributes (access or cache)
P4HWVIRT_E_HYP_MAP_INCONSISTENT	0x10012: Error while walking page tables. This error can be raised if there is an error during a page table walk. This usually mean that the page tables have been altered or that there is some inconsistency in the KMEM mapping.
P4HWVIRT_E_HYP_MAP_GIC_OVERRIDE	0x10013: GIC address override during a mapping. This error is raised if a mapping request for a guest is overriding the GIC configuration. That can be mapping the real GIC hardware or trying to map something where the GIC Distributor should be emulated.
P4HWVIRT_E_HYP_IRQ_NOT_GRANTED	0x10014: Cannot forward an interrupt if not granted in user application. The user application tried to forward an interrupt to the guest and the given interrupt number was not granted to the calling user application. Only interrupt granted to the calling task can be forwarded to a guest.
P4HWVIRT_E_HYP_IRQ_INVALID_IRQID	0x10015: A call with an IRQ ID parameter was done with an invalid interrupt number. This is because the interrupt number is greater then the number of supported interrupts. For a request to forward an interrupt, this can be due to an attempt to forward a per core interrupt (<32).

P4HWVIRT_E_HYP_IRQ_CANNOT_ATTACH	0x10016: A request to forward an interrupt was done and it is not possible to attach to the wanted interrupt. This usually mean that something else in PikeOS is already attached to the interrupt number.
P4HWVIRT_E_HYP_GUEST_INVALID_CORE	0x10017: Guest is assigned to a core on which hardware virtualization was not initialised. One of the cores to on which the guest should be executed has no proper hardware virtualization support. This usually means that the PSP was not started in hypervisor mode on one of the cores which was assigned to the guest (ie one of the core activated in the manager partition cpumask). This error is also raised if an operation on an invalid core is requested (interrupt generation or run for example). You should check if your bootloader is properly starting PikeOS in hypervisor mode on all the cores of your target.
P4HWVIRT_E_HYP_EXTERNAL_HANDLER_INIT	0x10018: Error during an external exception handler init. An error was raised during an external exception handler init for the current guest. Depending on the external handlers you have included in your configuration, you should check the documentation provided with them and perhaps activate some verbose mode on them to have more information on the error.
P4HWVIRT_E_HYP_CANCELED	0x10019: The current core was deleted. This can happen during a partition reboot or if the thread calling the hypervisor has been canceled or deleted.
P4HWVIRT_E_HYP_INIT_STATE	0x1001a: The hypervisor is not in the right init state This error can be returned to external exception handlers if a function is called during the wrong init state or if the hypervisor is deactivated in the configuration.
P4HWVIRT_E_HYP_INVAL_EXCEPT	0x1001b: Invalid exception ID The exception ID passed as parameter is not a valid exception number supported by the hypervisor.
P4HWVIRT_E_HYP_INVAL_NULL	0x1001c: Invalid NULL pointer One of the passed parameter has an invalid value NULL value.
P4HWVIRT_E_HYP_INVAL_GUESTID	0x1001d: Invalid guest ID The guest ID given as parameter is not valid.

Enumeration type *p4hwvirt_e_mng_cfg_t*

Manager configuration errors (Group 0x20000)

This is listing all errors that can be returned by the manager during configuration.

Name	**Description**

P4HWVIRT_E_MNG_CFG_GROUP	0x20000: Configuration module error in the manager. This error happens when a configuration problem is detected. This error is raised as default module error.
P4HWVIRT_E_MNG_CFG_OPEN_PROP	0x20001: Manager configuration open property error. This error happens during manager configuration (initialization) when the manager try to open a property.
P4HWVIRT_E_MNG_CFG_READ_PROP	0x20002: Manager configuration read property error. This error happens during manager configuration (initialization) when the manager try to read a property.
P4HWVIRT_E_MNG_CFG_CLOSE_PROP	0x20003: Manager configuration close property error. This error happens during manager configuration (initialization) when the manager try to close a property.

Enumeration type *p4hwvirt_e_mng_dio_t*

Manager Direct IO errors (Group 0x30000)

This is listing all errors that can be returned by the manager in Direct IO module.

Name	Description
P4HWVIRT_E_MNG_DIO_GROUP	0x30000: Direct IO module error in the manager. This error happens when a problem is detected in Direct IO module. This error is raised as default module error.
P4HWVIRT_E_MNG_DIO_INIT	0x30001: Direct IO initialization error. This error happens when a problem is detected during initialization in Direct IO module.
P4HWVIRT_E_MNG_DIO_GRANT_IRQ	0x30002: Grant IRQ Direct IO error. This error happens when granting IRQ for Direct IO fails.
P4HWVIRT_E_MNG_DIO_FWD_IRQ	0x30003: Forward IRQ Direct IO error. This error happens when forwarding IRQ for Direct IO fails.
P4HWVIRT_E_MNG_DIO_MEM_MAP	0x30004: Memory Map Direct IO error. This error happens when mapping memory for Direct IO fails.
P4HWVIRT_E_MNG_DIO_IOMEM_MAP	0x30005: IO Memory Map Direct IO error. This error happens when mapping IO memory for Direct IO fails.
P4HWVIRT_E_MNG_DIO_GRANT_DEV	0x30006: Grant guest access error. Could not grant the access to the guest for a DirectIO device.

Enumeration type *p4hwvirt_e_mng_fdt_t*

Manager FDT errors (Group 0x40000)

This is listing all errors that can be returned by the manager in FDT module.

Name	Description
P4HWVIRT_E_MNG_FDT_GROUP	0x40000: FDT module error in the manager. This error happens when a problem is detected in FDT module. This error is raised as default module error.
P4HWVIRT_E_MNG_FDT_DTB_HEADER	0x40001: DTB header error. This error happens when an error in DTB header is detected in FDT module.
P4HWVIRT_E_MNG_FDT_DTB_INCREASE_SIZE	0x40002: DTB increase size error. This error happens when trying to increase DTB size fails in FDT module.

Enumeration type *p4hwvirt_e_mng_dtb_t*

Manager DTB errors (Group 0x50000)

This is listing all errors that can be returned by the manager in DTB module.

Name	Description
P4HWVIRT_E_MNG_DTB_GROUP	0x50000: DTB module error in the manager. This error happens when a problem is detected in DTB module. This error is raised as default module error.
P4HWVIRT_E_MNG_DTB_INIT	0x50001: DTB initialization error. This error happens during DTB initialization.
P4HWVIRT_E_MNG_DTB_FINALIZE	0x50002: DTB finalization error. This error happens during DTB finalization.
P4HWVIRT_E_MNG_DTB_OFFSET	0x50003: DTB offset error. DTB offset is out of memory.

Enumeration type *p4hwvirt_e_mng_guest_t*

Manager guest errors (Group 0x60000)

This is listing all guest errors in the manager.

Name	Description
P4HWVIRT_E_MNG_GUEST_GROUP	0x60000: Guest error in the manager. This error happens when a problem is detected by the guest. This error is raised as default module error.

P4HWVIRT_E_MNG_GUEST_INIT	0x60001: Guest initialization error. This error happens during guest initialization.
P4HWVIRT_E_MNG_GUEST_START	0x60002: Start guest error. This error happens when start guest fails.
P4HWVIRT_E_MNG_GUEST_NOT_RUNNING	0x60003: Guest not running error. This error happens when guest is not running.
P4HWVIRT_E_MNG_GUEST_CREATE	0x60004: Guest creation error. Error creating the guest.
P4HWVIRT_E_MNG_GUEST_GET_RAM	0x60005: Get RAM error. Error getting guest RAM memory.
P4HWVIRT_E_MNG_GUEST_GET_MEM	0x60006: Get guest memory error. Error getting guest memory information.
P4HWVIRT_E_MNG_GUEST_VMEM_ALLOC	0x60007: Allocate virtual memory error. Not enough virtual address space.
P4HWVIRT_E_MNG_GUEST_VMEM_POOL_ALLOC	0x60008: Allocate virtual memory error. Cannot map guest memory in manager, may need more KMEM.
P4HWVIRT_E_MNG_GUEST_MAP_MEM	0x60009: Guest memory mapping error. Error mapping memory to the guest.
P4HWVIRT_E_MNG_GUEST_UNMAP_MEM	0x6000A: Guest memory unmapping error. Cannot unmap guest memory section from the manager.
P4HWVIRT_E_MNG_GUEST_START_ADDR_OFF	0x6000B: Guest start address offset error. Error getting guest start address from offset.
P4HWVIRT_E_MNG_GUEST_START_ADDR_IPA	0x6000C: Converting guest start address to the IPA error. Error converting the guest start address to the IPA.
P4HWVIRT_E_MNG_GUEST_START_CORE	0x6000D: Start guest error. Error starting the guest core.
P4HWVIRT_E_MNG_GUEST_STOP_CORE	0x6000E: Stop guest error. Error stopping the guest core.
P4HWVIRT_E_MNG_GUEST_MAPPING_REQ	0x6000F: Invalid mapping request. This error happens when the mapping request is not valid.

Enumeration type *p4hwvirt_e_mng_vmm_wtd_t*

VMM watchdog errors (Group 0x70000)

This is listing all errors that can be returned by VMM watchdog module.

Name	Description
P4HWVIRT_E_VMM_WTD_GROUP	0x70000: VMM watchdog module error in the manager. This error happens when a problem is detected in VMM watchdog module. This error is raised as default module error.
P4HWVIRT_E_VMM_WTD_TIMEOUT	0x70001: VMM watchdog error. This error happens when VMM Watchdog is not refreshed by the guest.
P4HWVIRT_E_VMM_WTD_BAD_TIMEOUT	0x70002: VMM watchdog bad timeout error. This error happens when VMM Watchdog has a bad timeout configured.
P4HWVIRT_E_VMM_WTD_CANCEL	0x70003: VMM watchdog thread canceled error. This error happens when VMM Watchdog thread was canceled.

Enumeration type *p4hwvirt_e_mng_vmm_t*

VMM manager errors (Group 0x80000)

This is listing all errors that can be returned by the VMM module.

Name	Description
P4HWVIRT_E_MNG_VMM_GROUP	0x80000: VMM module error in the manager. This error happens when a problem is detected in manager. This error is raised as default module error.
P4HWVIRT_E_MNG_VMM_DTB_INIT	0x80001: VMM specific DTB entries initialization error. Cannot initialize VMM specific DTB entries.
P4HWVIRT_E_MNG_VMM_DRV_HANDLER	0x80002: VMM driver handler error. The current VMM driver has no handler.
P4HWVIRT_E_MNG_VMM_DRV_DTB	0x80003: Error adding the DTB node. Error adding the DTB node for the VMM driver.
P4HWVIRT_E_MNG_VMM_DRV_INIT	0x80004: VMM driver initialization error. Error during the VMM driver initialization.
P4HWVIRT_E_MNG_VMM_DRV_NOENT	0x80005: Non existing driver error. VMM Message for non-existing driver received.

P4HWVIRT_E_MNG_VMM_INFO	0x80006: VMM INFO Unknown operation error. VMM_DEV_INFO: Unknown operation.
P4HWVIRT_E_MNG_VMM_VMAPI	0x80007: VMM VMAPI Unkown operation error. VMM_DEV_VMAPI: Unkown operation.

Enumeration type *p4hwvirt_e_mng_p4bus_t*

P4bus manager errors (Group 0x90000)

This is listing all errors that can be returned by the P4bus module.

Name	Description
P4HWVIRT_E_MNG_P4BUS_GROUP	0x90000: P4bus module error in the manager. This error happens when a problem is detected in manager. This error is raised as default module error.
P4HWVIRT_E_MNG_P4BUS_INVAL_DEV	0x90001: P4BUS invalid device error. P4BUS operation with invalid device.
P4HWVIRT_E_MNG_P4BUS_INVAL_ADDR	0x90002: P4BUS invalid address error. P4BUS operation with invalid address.
P4HWVIRT_E_MNG_P4BUS_MAX_DEV	0x90003: P4BUS Max devices error. Too many P4BUS devices.
P4HWVIRT_E_MNG_P4BUS_RESOURCES	0x90004: P4BUS resources error. Cannot allocate our internal data for the P4BUS initialization.
P4HWVIRT_E_MNG_P4BUS_DRV_INIT	0x90005: P4BUS driver initialization error. Error during the P4BUS driver initialization.
P4HWVIRT_E_MNG_P4BUS_DEV_INIT	0x90006: Register P4BUS device error. Error registering the P4BUS device.
P4HWVIRT_E_MNG_P4BUS_DTB	0x90007: Add P4BUS device error. Error adding P4Bus device to DTB.
P4HWVIRT_E_MNG_P4BUS_THREAD	0x90008: P4BUS ioring thread creation error. Error creating p4bus ioring thread.
P4HWVIRT_E_MNG_P4BUS_INVAL_OP_STATUS	0x90009: Bad operation status. This error happens when the operation status is different of ready.

Enumeration type *p4hwvirt_e_mng_cpu_t*

CPU manager errors (Group 0xA0000)

This is listing all CPU errors that can be returned by the manager.

Name	Description
P4HWVIRT_E_MNG_CPU_GROUP	0xA0000: CPU module error in the manager. This error happens when a problem is detected in manager. This error is raised as default module error.
P4HWVIRT_E_MNG_CPU_NB_REAL_CORE	0xA0001: Real core for guest unavailable. The number of cores requested by the guest is lower than the real number of cores. Cannot move the guest core to its real core.
P4HWVIRT_E_MNG_CPU_CORE_AFF	0xA0002: Set core affinity error. Cannot set the guest core thread affinity.
P4HWVIRT_E_MNG_CPU_HVC	0xA0003: Invalid HVC error. Guest called hvc with an invalid number.
P4HWVIRT_E_MNG_CPU_CP15	0xA0004: Invalid CP15 access error. Invalid CP15 access from the guest.
P4HWVIRT_E_MNG_CPU_CPU_REGS	0xA0005: Invalid CPU register access error. Invalid CPU register access from the guest.
P4HWVIRT_E_MNG_CPU_IOACCESS	0xA0006: Invalid data address error. Guest tried to access data at invalid address.
P4HWVIRT_E_MNG_CPU_ASYNC_ABORT	0xA0007: Asynchronous abort error. Asynchronous abort from the guest.
P4HWVIRT_E_MNG_CPU_FPU	0xA0008: FPU usage error. Forbidden usage of the FPU.
P4HWVIRT_E_MNG_CPU_SMC	0xA0009: SMC instruction error. Forbidden SMC instruction.
P4HWVIRT_E_MNG_CPU_EXCEPT	0xA000A: Unhandled exception error. Unhandled exception received.
P4HWVIRT_E_MNG_CPU_PREFETCH	0xA000B: Execution code invalid address error. Guest tried to execute code at invalid address.

Enumeration type *p4hwvirt_e_mng_gen_t*

Generic manager errors (Group 0xB0000)

This is listing all generic errors that can be returned by the manager.

Name	Description
P4HWVIRT_E_MNG_GEN_GROUP	0xB0000: Generic error in the manager. This error happens when a problem is detected in manager. This error is raised as default module error.
P4HWVIRT_E_MNG_GEN_KDEV_GRANT	0xB0001: Error reading kernel device from properties. This error happens when reading kernel device from properties fails.
P4HWVIRT_E_MNG_GEN_OPEN_CMDLINE	0xB0002: Open command line error. This error happens when opening command line file fails.
P4HWVIRT_E_MNG_GEN_READ_CMDLINE	0xB0003: Read command line error. This error happens when reading command line from file fails.
P4HWVIRT_E_MNG_GEN_CLOSE_CMDLINE	0xB0004: Close command line error. This error happens when closing command line from file fails.
P4HWVIRT_E_MNG_GEN_TASK_ATTR	0xB0005: Get task attributes error. This error happens when getting our task attributes fails.
P4HWVIRT_E_MNG_GEN_GET_MEM	0xB0006: Get memory error. This error happens when getting manager memory fails.
P4HWVIRT_E_MNG_GEN_STAT_MEM	0xB0007: Get memory information error. This error happens when getting memory information fails.
P4HWVIRT_E_MNG_GEN_VMEM_ALLOC	0xB0008: VM allocation error. Not enough virtual memory to map the manager memory.
P4HWVIRT_E_MNG_GEN_VMEM_POOL_ALLOC	0xB0009: VM pool allocation error. This error happens when mapping manager memory fails, there is not enough KMEM.
P4HWVIRT_E_MNG_GEN_MEM_ALLOC_HYP	0xB000A: Hypervisor memory allocation error. This error happens when there is not enough hypervisor memory for allocation.
P4HWVIRT_E_MNG_GEN_GET_PHYS_HYP	0xB000B: Get physical hypervisor address error. This error happens when getting physical address of hypervisor memory fails.
P4HWVIRT_E_MNG_GEN_MEM_ALLOC_THREAD	0xB000C: Allocate memory for thread error. This error happens when allocation for thread cannot be done.

P4HWVIRT_E_MNG_GEN_GET_HYP_MEM	0xB000D: Get hypervisor memory error. This error happens when hypervisor memory allocation fails.
P4HWVIRT_E_MNG_GEN_INVALID_MMAP	0xB000E: Invalid mampping error. Error, the mapping request is not valid.
P4HWVIRT_E_MNG_GEN_MEM_USED	0xB000F: Allocation out of memory area error. Someone allocated memory out of the current area.
P4HWVIRT_E_MNG_GEN_MEM_NOT_POOL	0xB0010: Memory area property error. The memory area is not defined as a POOL.
P4HWVIRT_E_MNG_GEN_OPEN_FILE	0xB0011: Open file error. Cannot open the file.
P4HWVIRT_E_MNG_GEN_READ_FILE	0xB0012: Read file error. Cannot read the file.
P4HWVIRT_E_MNG_GEN_MAP_FILE	0xB0013: Map file error. Cannot map the file.
P4HWVIRT_E_MNG_GEN_CLOSE_FILE	0xB0014: Close file error. Cannot map the file.
P4HWVIRT_E_MNG_GEN_GET_INFO_FILE	0xB0015: Get information file error. Error getting the file information.
P4HWVIRT_E_MNG_GEN_FILE_TO_GUEST_OFF	0xB0016: Guest file offset error. The file cannot be copied to the guest memory at the requested offset.
P4HWVIRT_E_MNG_GEN_FILE_TO_GUEST_SIZE	0xB0017: Guest file too big error. The file is to big for the guest memory.
P4HWVIRT_E_MNG_GEN_CACHE_FLUSH	0xB0018: Flush cache error. Error flushing cache.
P4HWVIRT_E_MNG_GEN_REG_OFFSET	0xB0019: Register offset error. Register offset is out of memory.
P4HWVIRT_E_MNG_GEN_CRE-ATE_CORE_THREAD	0xB001A: Create core thread error. Cannot create guest core thread.
P4HWVIRT_E_MNG_GEN_VIRTIO_INIT	0xB001B: Virtio module intialization error. Error during the initialization of virtio module.

P4HWVIRT_E_MNG_GEN_UNKNOWN	0xB001C: Error unknown. This error happens when the error is unknown.

14.2 HWVIRT Global Interface

This section describes all types and definitions common to User, KDEV and PSP interface. This is mainly the definition of guest exceptions and guest context.

In the next description, an exception handler can be a user application or a KDEV driver. Exiting an exception handler means calling P4HWVIRT_GUEST_RUN from a user application.

For each exception the Input behaviour is describing what is done by the hypervisor when a guest is started with the exception structure exception_type field set to the given exception number.

The following is always done when exiting from an exception handler:

- the core context is updated from the given context if run_flags P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT bit is set.
- the core PC is increased of one instruction if run_flags P4HWVIRT_RUNFLAGS_ADVANCE_PC bit is set.

The output behaviour is describing when the given exception can occur, what is done internally and what is done with the exception structure and context before calling an exception handler.

The following is always done before calling an exception handler:

- if P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT bit is set in run_flags, the context passed by the user application is updated with the current context of the guest.

Depending on the exception_type, some extra information might be available in the exception structure.

For each exception raised a preemption point is called in the hypervisor, which means that a rescheduling to an other thread of the system can occur when the exception is raised by the guest or the system.

When the exception is handled by an external exception handler, the context passed is directly the one of the guest and can be modified directly by the handler. As a consequence the P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT bit is not taken into account, only the P4HWVIRT_RUNFLAGS_ADVANCE_PC will advance the PC when set.

14.2.1 Structure Definitions

14.2.1.1 struct P4hwvirt_exception_ioerror_s

Description of a guest IO error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_IO occurs.

When handling IO errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

Synopsis:

```
struct P4hwvirt_exception_ioerror_s {
    P4_phys_addr_t addr;
    P4_address_t vaddr;
    P4hwvirt_dir_t dir;
    P4hwvirt_size_t size;
    P4_uint64_t value;
};
```

Structure Element Description:

addr Physical address accessed

vaddr Guest virtual address accessed

dir Direction of the access

size Size of the access

value Value writen for write access, value to give back as result for read access.

Associated Data Type

P4hwvirt_exception_ioerror_t Description of a guest IO error.

14.2.1.2 struct P4hwvirt_exception_vmmcall_s

Description of a guest VMM exception.

This structure is giving the content of the message sent by a guest for a VMM Bus call and is filled when an exception P4HWVIRT_EXCEPT_VMMCALL occurs.

The content of the message must be updated with the answer to give back to the guest when returning from this exception.

Synopsis:

```
struct P4hwvirt_exception_vmmcall_s {
    P4_cpureg_t message[VMM_MESSAGE_SIZE];
};
```

Structure Element Description:

message VMM message data

Associated Data Type

P4hwvirt_exception_vmmcall_t Description of a guest VMM exception.

14.2.1.3 struct P4hwvirt_exception_hypcall_s

Description of an hypervisor call exception.

When a P4HWVIRT_EXCEPT_HYPCALL exception occurs, this is giving the extra ID of the hypervisor call done by the guest.

Synopsis:

```
struct P4hwvirt_exception_hypcall_s {
    P4_cpureg_t id;
};
```

Structure Element Description:

id Hypervisor call identifier

Associated Data Type

P4hwvirt_exception_hypcall_t Description of an hypervisor call exception.

14.2.1.4 struct P4hwvirt_exception_unsupp_s

Description of an unsupported exception.

When a P4HWVIRT_EXCEPT_UNSUPP exception occurs, this can be filled by the architecture with some extra information on the exception that occured.

Check the architecture specific manual for more information.

Synopsis:

```
struct P4hwvirt_exception_unsupp_s {
    P4_cpureg_t arch_info[4];
};
```

Structure Element Description:

arch_info extra architecture info when unsupported

Associated Data Type

P4hwvirt_exception_unsupp_t Description of an unsupported exception.

14.2.1.5 struct P4hwvirt_exception_core_start_s

Description of a core start exception.

This structure contains the PC at which to start a secondary core and the number of secondary core to start when an external exception handler wants the user application to start a secondary core.

Synopsis:

```
struct P4hwvirt_exception_core_start_s {
    P4_cpureg_t pc;
    P4_cpuid_t coreid;
    P4_uint32_t padding;
};
```

Structure Element Description:

pc Address at which to start the given core.

coreid Which core to start

padding unused padding

Associated Data Type

P4hwvirt_exception_core_start_t Description of a core start exception.

14.2.1.6 struct P4hwvirt_exception_core_stop_s

Description of a core stop exception.

This structure contains the number of the secondary core to stop when an external handler want the user application to stop a secondary core.

Synopsis:

```
struct P4hwvirt_exception_core_stop_s {
    P4_cpuid_t coreid;
    P4_uint32_t padding;
};
```

Structure Element Description:

coreid Which core to stop

padding unused padding

Associated Data Type

P4hwvirt_exception_core_stop_t Description of a core stop exception.

14.2.1.7 struct P4hwvirt_exception_s

Guest exception information.

Contains the exception description and information and action to be performed when returning to guest execution (depending on the type of exception)

Depending on the exception type, one of the P4hwvirt_exception_* might contain some extra information.

Note:

The P4hwvirt_exception_* are grouped in one anonymous union.

Synopsis:

```
struct P4hwvirt_exception_s {
    P4_cpureg_t exception_type;
    P4_cpureg_t pc;
    P4_cpureg_t esr;
    P4_cpureg_t run_flags;
    P4hwvirt_exception_vmmcall_t vmmcall;
    P4hwvirt_exception_hypcall_t hypcall;
    P4hwvirt_exception_ioerror_t io;
    P4hwvirt_exception_unsupp_t unsupp;
    P4hwvirt_exception_core_start_t start;
    P4hwvirt_exception_core_start_t stop;
    P4hwvirt_exception_arch_t hw_arch;
};
```

Structure Element Description:

exception_type Type of the exception (one of *P4HWVIRT_EXCEPT_** (see section 14.2.2))

pc Address of execution when the exception occured (virtual)

esr Real value of the syndrome register

run_flags Flags to define action to be done before reexecuting the guest. This must be set using *P4HWVIRT_RUNFLAGS_** (see section 14.2.2) bits (several possible).

vmmcall Information for a *P4HWVIRT_EXCEPT_VMMCALL* (see section 14.2.2) exception

hypcall Information for a *P4HWVIRT_EXCEPT_HYPCALL* (see section 14.2.2) exception

io Information for a *P4HWVIRT_EXCEPT_IO* (see section 14.2.2) exception

unsupp Information for a *P4HWVIRT_EXCEPT_UNSUPP* (see section 14.2.2) exception

start Information for a P4HWVIRT_CORE_START exception

stop Information for a P4HWVIRT_CORE_STOP exception

hw_arch Information for architecture specific exceptions

Associated Data Type

P4hwvirt_exception_t Guest exception information.

14.2.2 Defines

P4HWVIRT_EXCEPT_NONE

No exception guest exception.

Description:

Output behaviour:

- happens when an hardware interrupt occurs during guest execution
- execution is restarted after a preemption point in the kernel
- not possible to get in user application

Input behaviour:

- no specific action

P4HWVIRT_EXCEPT_HYPCALL

Hypervisor call exception.

Description:

Output behaviour:

- happens when a guest is executing the hypervisor call instruction
- exception hypcall structure is filled
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- no specific action

P4HWVIRT_EXCEPT_IO

IO exception.

Description:

Output behaviour:

- happens when a guest accessed an invalid address (usually a data abort or an instruction abort)
- exception io structure is filled
- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- access to interrupt controller is handled internally

Input behaviour:

- for a read IO exception, the proper register is updated with the value field of the io structure.

P4HWVIRT_EXCEPT_CANCEL

Internal error exception.

Description:

This exception happens when an internal error is occuring during a guest execution, which usually means the thread running the core has been deleted.

Output behaviour:

- happens when the thread we are running on is deleted
- no action is done, the hypervisor exits directly

Input behaviour:

- handled as a NONE exception

P4HWVIRT_EXCEPT_UNSUPP

Unsupported exception.

Description:

Output behaviour:

- happens when an unsupported exception is generated by the guest

- some architecture specific information might be available in the exception
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- no internal handling

Input behaviour:

- handled as a NONE exception

P4HWVIRT_EXCEPT_VMMCALL

Virtual Machine Monitor exception.

Description:

Output behaviour:

- happens when the guest is executing the hypervisor call for the VMM Bus
- exception vmmcall structure is filled
- no internal handling

Input behaviour:

- part of the context is updated using the vmmcall structure of the exception

P4HWVIRT_EXCEPT_CORE_START

Core start exception.

Description:

Output behaviour:

- this exception cannot be generated directly by the guest, it is a virtual exception generated by an external exception handler to ask the user application to start a secondary core
- the start structure is filled
- no internal handling

Input behaviour:

- handled as a NONE exception

P4HWVIRT_EXCEPT_CORE_STOP

Core stop exception.

Description:

Output behaviour:

- this exception cannot be generated directly by the guest, it is a virtual exception generated by an external exception handler to ask the user application to stop a secondary core
- the stop structure is filled
- no internal handling

Input behaviour:

- handled as a NONE exception

P4HWVIRT_EXCEPT_REBOOT

Reboot exception.

Description:

Output behaviour:

- this exception cannot be generated directly by the guest, it is a virtual exception generated by an external exception handler to ask the user application to reboot the guest
- no internal handling

Input behaviour:

- handled as a NONE exception

P4HWVIRT_EXCEPT_POWEROFF

Power Off exception.

Description:

Output behaviour:

- this exception cannot be generated directly by the guest, it is a virtual exception generated by an external exception handler to ask the user application to power off the guest
- no internal handling

Input behaviour:

- handled as a NONE exception

P4HWVIRT_EXCEPT_BREAK

Break exception.

Description:

Output behaviour:

- happens when the guest is executing a breakpoint instruction or when P4HWVIRT_GUEST_CORE_BREAK was called for the current guest core
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- handled as a NONE exception

P4HWVIRT_EXCEPT_ARCH

Beginning of architecture specific exceptions.

Description:

You can find architecture specific exceptions here:

- *ARM Specific exceptions* (see section 14.3.2)

P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT

Update context run flag.

Description:

Run preparation flags

When this bit is set in the exception run_flags, the guest core context is updated with the content of the context passed by the user application to the P4HWVIRT_GUEST_CORE_RUN before executing the guest.

P4HWVIRT_RUNFLAGS_ADVANCE_PC

Advance PC run flag.

Description:

When this bit is set in the exception run_flags, the guest execution pointer is advanced of one instruction before starting the guest when P4HWVIRT_GUEST_CORE_RUN is called.

14.2.3 Data Type Definitions

P4hwvirt_dir_t Guest access direction.

For IO errors, this is used to give the direction of the access (read, write or execute).

P4hwvirt_size_t Access size.

For IO errors, this is used to give the access size (8, 16, 32 or 64bit).

P4hwvirt_exception_ioerror_t Description of a guest IO error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_IO occurs.

When handling IO errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

P4hwvirt_exception_vmmcall_t Description of a guest VMM exception.

This structure is giving the content of the message sent by a guest for a VMM Bus call and is filled when an exception P4HWVIRT_EXCEPT_VMMCALL occurs.

The content of the message must be updated with the answer to give back to the guest when returning from this exception.

P4hwvirt_exception_hypcall_t Description of an hypervisor call exception.

When a P4HWVIRT_EXCEPT_HYPCALL exception occurs, this is giving the extra ID of the hypervisor call done by the guest.

P4hwvirt_exception_unsupp_t Description of an unsupported exception.

When a P4HWVIRT_EXCEPT_UNSUPP exception occurs, this can be filled by the architecture with some extra information on the exception that occured.

Check the architecture specific manual for more information.

P4hwvirt_exception_core_start_t Description of a core start exception.

This structure contains the PC at which to start a secondary core and the number of secondary core to start when an external exception handler wants the user application to start a secondary core.

P4hwvirt_exception_core_stop_t Description of a core stop exception.

This structure contains the number of the secondary core to stop when an external handler want the user application to stop a secondary core.

P4hwvirt_exception_t Guest exception information.

Contains the exception description and information and action to be performed when returning to guest execution (depending on the type of exception)

Depending on the exception type, one of the P4hwvirt_exception_* might contain some extra information.

Note:

The P4hwvirt_exception_* are grouped in one anonymous union.

14.2.4 Enumerations

Enumeration type *P4hwvirt_dir_e*

Guest access direction.

For IO errors, this is used to give the direction of the access (read, write or execute).

Name	Description
P4HWVIRT_ACCESS_READ	Read access
P4HWVIRT_ACCESS_WRITE	Write access
P4HWVIRT_ACCESS_EXEC	Execute access
P4HWVIRT_ACCESS_INVALID	Invalid or unhandled access

Enumeration type *P4hwvirt_size_e*

Access size.

For IO errors, this is used to give the access size (8, 16, 32 or 64bit).

Name	Description
P4HWVIRT_SIZE_8	8bit access
P4HWVIRT_SIZE_16	16bit access
P4HWVIRT_SIZE_32	32bit access
P4HWVIRT_SIZE_64	64bit access

14.3 HWVIRT Global Interface (ARM specific)

This section describes all global types and definitions specific to ARM architecture. Those are used by a user application managing a guest or by a kernel external exception handler.

14.3.1 Structure Definitions

14.3.1.1 struct P4hwvirt_exception_arm_cp15_32_s

Description of a guest CP15 32bit error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_CP15_32 occurs.

When handling CP15 errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

Synopsis:

```
struct P4hwvirt_exception_arm_cp15_32_s {
    P4_uint32_t opc1;
    P4_uint32_t opc2;
    P4_uint32_t crn;
    P4_uint32_t crm;
    P4_uint32_t dir;
    P4_uint32_t value;
};
```

Structure Element Description:

opc1 cp15 opc1 value

opc2 cp15 opc2 value

crn cp15 crn value

crm cp15 crm value

dir direction (P4HWVIRT_ACCESS_READ or P4HWVIRT_ACCESS_WRITE)

value value

Associated Data Type

P4hwvirt_exception_arm_cp15_32_t Description of a guest CP15 32bit error.

14.3.1.2 struct P4hwvirt_exception_arm_cp15_64_s

Description of a guest CP15 64bit error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_CP15_64 occurs.

When handling CP15 errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

Synopsis:

```
struct P4hwvirt_exception_arm_cp15_64_s {
    P4_uint32_t opc1;
    P4_uint32_t crm;
    P4_uint32_t dir;
    P4_uint32_t padding;
    P4_uint64_t value;
};
```

Structure Element Description:

opc1 cp15 opc1

crm cp15 crm

dir direction (P4HWVIRT_ACCESS_READ or P4HWVIRT_ACCESS_WRITE)

padding padding to align value to 8

value value

Associated Data Type

P4hwvirt_exception_arm_cp15_64_t Description of a guest CP15 64bit error.

14.3.1.3 struct P4hwvirt_exception_msr_mrs_s

Description of a guest MRS/MSR error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_MRS_MSR occurs.

When handling MRS/MSR errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

Synopsis:

```
struct P4hwvirt_exception_msr_mrs_s {
    P4_uint32_t op0;
    P4_uint32_t op1;
    P4_uint32_t op2;
    P4_uint32_t crn;
    P4_uint32_t crm;
    P4_uint32_t dir;
    P4_uint64_t value;
};
```

Structure Element Description:

op0 msr/mrs op0

op1 msr/mrs op1

op2 msr/mrs op2

crn msr/mrs crn

crm msr/mrs crm

dir msr/mrs direction (P4HWVIRT_ACCESS_READ or P4HWVIRT_ACCESS_WRITE)

value value

Associated Data Type

P4hwvirt_exception_msr_mrs_t Description of a guest MRS/MSR error.

14.3.1.4 union P4hwvirt_exception_arch_u

ARM specific exception information.

Synopsis:

```
union P4hwvirt_exception_arch_u {
    P4hwvirt_exception_arm_cp15_32_t cp15_32;
    P4hwvirt_exception_arm_cp15_64_t cp15_64;
    P4hwvirt_exception_msr_mrs_t msr_mrs;
```

```
};
```

Union Element Description:

cp15_32 Information for a *P4HWVIRT_EXCEPT_CP15_32* (see section 14.3.2) exception

cp15_64 Information for a *P4HWVIRT_EXCEPT_CP15_64* (see section 14.3.2) exception

msr_mrs Information for a *P4HWVIRT_EXCEPT_MRS_MSR* (see section 14.3.2) exception

Associated Data Type

P4hwvirt_exception_arch_t ARM specific exception information.

14.3.2 Defines

HYP_64BIT_SUPPORT
This is defined when the architecture supports running 64bit guests.

HYP_32BIT_SUPPORT
This is defined when the architecture supports running 32bit guests.

P4HWVIRT_NUM_REGS
Number of registers in a guest context.

P4HWVIRT_NUM_USER_REGS
Number of user registers in a guest context.

P4HWVIRT_REG_SP
Stack pointer register index in a guest context.

P4HWVIRT_REG_LR
Link register index in a guest context.

P4HWVIRT_REG_PC
Program Counter index in a guest context.

P4HWVIRT32_NUM_REGS
Number of registers in a 32bit guest context.

P4HWVIRT32_NUM_USER_REGS
Number of user registers in a 32bit guest context.

P4HWVIRT32_REG_SP
Stack pointer register index in a 32bit guest context.

P4HWVIRT32_REG_LR
Link register index in a 32bit guest context.

P4HWVIRT32_REG_PC
Program Counter index in a 32bit guest context.

P4HWVIRT_EXCEPT_SMC
SMC guest exception.

Description:

Output behaviour:

- happens when a guest is executing the smc instruction
- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- no specific action

P4HWVIRT_EXCEPT_CP15_32

32bit CP15 guest exception

Description:

Output behaviour:

- happens when a guest is executing a forbidden cp15 32bit instruction
- exception hw_arch.cp15_32 is filled
- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- in case of a read cp15, the context is updated with the value of the exception hw_arch.cp15_32 structure.

P4HWVIRT_EXCEPT_CP15_64

64bit CP15 guest exception

Description:

Output behaviour:

- happens when a guest is executing a forbidden cp15 64bit instruction
- exception hw_arch.cp15_64 is filled
- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- in case of a read cp15, the context is updated with the value of the exception hw_arch.cp15_32 structure.

P4HWVIRT_EXCEPT_MRS_MSR

MRS or MSR guest exception.

Description:

Output behaviour:

- happens when a guest is executing a forbidden msr or mrs instruction
- exception hw_arch.msr_mrs is filled
- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- in case of an MRS (read), the context is updated with the value of the exception hw_arch.msr_mrs structure.

P4HWVIRT_EXCEPT_FPU

FPU exception.

Description:

Output behaviour:

- happens when a guest is accessing the FPU when it was not allowed

- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- no specific action

P4HWVIRT_EXCEPT_ASYNC

Asynchronous exception.

Description:

Output behaviour:

- happens when a guest is doing an hardware access generating an asynchronous exception
- P4HWVIRT_RUNFLAGS_ADVANCE_PC is set
- P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT is set
- no internal handling

Input behaviour:

- no specific action

14.3.3 Data Type Definitions

P4hwvirt_exception_arm_cp15_32_t Description of a guest CP15 32bit error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_CP15_32 occurs.

When handling CP15 errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

P4hwvirt_exception_arm_cp15_64_t Description of a guest CP15 64bit error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_CP15_64 occurs.

When handling CP15 errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

P4hwvirt_exception_msr_mrs_t Description of a guest MRS/MSR error.

This structure is giving all the detailed information when an exception P4HWVIRT_EXCEPT_MRS_MSR occurs.

When handling MRS/MSR errors to simulate an access, in case of a read, the value field must be set to the value to be returned to the guest.

P4hwvirt_exception_arch_t ARM specific exception information.

14.4 HWVIRT User Host Interface

This section describes the communication API and types required to use the hypervisor from a user application to manage a guest.

This is the API used by the Manager to communicate with the hypervisor.

14.4.1 Structure Definitions

14.4.1.1 struct P4hwvirt_guest_s

Guest declaration structure.

Structure to declare a guest used during the P4HWVIRT_GUEST_INIT.

Some fields are to be set by the application calling P4HWVIRT_GUEST_INIT and some other will be set by the hypervisor during P4HWVIRT_GUEST_INIT and available for the application in the structure when returning from the call.

Synopsis:

```
struct P4hwvirt_guest_s {
    P4_cpureg_t features;
    P4_cpumask_t core_map;
    P4hwvirt_guest_arch_t hw_arch;
};
```

Structure Element Description:

features guest set of features

This is a set of bits representing some features that must be supported/activated for the guest. The architecture can defined specific flags here.

Those must be set by the calling application before calling P4HWVIRT_GUEST_INIT.

core_map map of the cores used by the guest.

Each physical core ID bit used by the guest must be set by the calling application in this map.

The hypervisor will configure everything so that core ID x of the guest is the x.th bit set in this map.

Unwanted behaviour could occur if the P4HWVIRT_GUEST_RUN is not called with the proper core ID on the right core according to this mapping (no interrupt generated, invalid execution, ..etc).

hw_arch Architecture parameters.

Architecture specific parameters.

Check *P4hwvirt_guest_arch_t* (see section 14.5.3) for your architecture.

Associated Data Type

P4hwvirt_guest_t Guest declaration structure.

14.4.1.2 struct P4hwvirt_operation_map_s

Structure used for a P4HWVIRT_CMD_GUEST_MAP ioctl.

Parameters to map a region to a guest.

Synopsis:

```
struct P4hwvirt_operation_map_s {
    P4_phys_addr_t dest_addr;
```

```
    P4_address_t virt_addr;
    P4_size_t size;
};
```

Structure Element Description:

dest_addr Guest Physical address (IPA) at which the area will be mapped

virt_addr Calling application virtual address of the area to be mapped

size Size of the area to be mapped

Associated Data Type

P4hwvirt_operation_map_t Structure used for a P4HWVIRT_CMD_GUEST_MAP ioctl.

14.4.1.3 struct P4hwvirt_operation_fw_irq_s

Structure used for a P4HWVIRT_CMD_GUEST_FORWARD_IRQ ioctl.

Parameters to forward and interrupt to a guest

Synopsis:

```
struct P4hwvirt_operation_fw_irq_s {
    P4_intid_t src_irq;
    P4_intid_t dest_irq;
};
```

Structure Element Description:

src_irq Host interrupt number to be forwarded to the guest

dest_irq Guest interrupt number to be generated

Associated Data Type

P4hwvirt_operation_fw_irq_t Structure used for a P4HWVIRT_CMD_GUEST_FORWARD_IRQ ioctl.

14.4.1.4 struct P4hwvirt_operation_gen_irq_s

Structure used for a P4HWVIRT_CMD_GUEST_GEN_IRQ ioctl.

Paramters to generate an interrupt to a guest

Synopsis:

```
struct P4hwvirt_operation_gen_irq_s {
    P4_intid_t dest_irq;
    P4_cpuid_t coreid;
};
```

Structure Element Description:

dest_irq Guest interrupt number to generate

coreid Core on which the interrupt must be generated (for PPIs only)

Associated Data Type

P4hwvirt_operation_gen_irq_t Structure used for a P4HWVIRT_CMD_GUEST_GEN_IRQ ioctl.

14.4.1.5 struct P4hwvirt_operation_run_s

Structure used for a P4HWVIRT_CMD_GUEST_CORE_RUN ioctl.

Input and output parameters to run a guest. The context and except content are used to execute the guest in input and are containing when done the information on the reason why the guest was stopped (exception and context)

Synopsis:

```
struct P4hwvirt_operation_run_s {
    P4hwvirt_exception_t except;
    P4_address_t context;
    P4_cpureg_t unused;
};
```

Structure Element Description:

except Exception information (input and output)

context Pointer to a table where the core context is taken and stored. The table must have P4HWVIRT_NUM_REGS P4_cpureg_t entries.

unused Unused field for padding

Associated Data Type

P4hwvirt_operation_run_t Structure used for a P4HWVIRT_CMD_GUEST_CORE_RUN ioctl.

14.4.2 Defines

P4HWVIRT_GUEST_FEATURE_32BIT

Guest 32bit feature flags.

Description:

This flag can be set by the user application in the guest structure feature on 64bit systems to create a 32bit guest. The flag is ignored on 32bit systems.

P4HWVIRT_GUEST_FEATURE_ARCH_BIT

Start of architecture specific feature flags.

P4HWVIRT_CMD_GUEST_INIT

Guest init IOCTL.

Description:

Initialise an hardware virtualized guest.

This function must be called once to initialized the guest.

Parameters:

P4hwvirt_guest_t Guest structure description. Some fields will be modified or set by the call.

Returns:

Upon success this will return P4_E_OK.

In case of error the following error codes can be returned:

P4HWVIRT_E_HYP_INVALID_GUEST_STATE Guest is already initialised

P4HWVIRT_E_HYP_IOCTL_INVALID_ARGUMENT The passed argument has not a size corresponding to a P4hwvirt_guest_t

P4HWVIRT_E_HYP_IOCTL_INVALID_USER_ADDRESS The passed argument is not a valid address in the caller address space (invalid pointer).

P4HWVIRT_E_HYP_GUEST_INVALID_CORE The core map in the passed guest structure contains one or several cores for which hardware virtualization was not properly initialized.

P4HWVIRT_E_HYP_NO_HYPMEM There is not enough hypervisor memory to create the guest.

P4HWVIRT_E_HYP_EXTERNAL_HANDLER_INIT An external exception handler did not properly initialised for this guest.

P4HWVIRT_E_HYP_MAP_OVERMAP The initialization is overriding part of an already existing mapping.

P4HWVIRT_E_HYP_MAP_INCONSISTENT An inconsistency in the page table was found (invalid pointer or flags in existing mappings).

P4HWVIRT_E_HYP_MAP_INVALID_ADDRESS The given address and size are not aligned to a page or if an address is not in the range of supported addresses (for example higher then 36bit hardware address on 32bit systems).

P4HWVIRT_E_HYP_MAP_GIC_OVERRIDE The given physical area contains one of the hardware GIC registers.

P4HWVIRT_CMD_GUEST_MAP

Map memory cacheable.

Description:

Map an area to a guest. The access attributes of the caller are kept in the guest.

The area size must be a multiple of P4_PAGESIZE and the addresses must be aligned to P4_PAGESIZE.

Parameters:

arg1 Application virtual address of the area to map.

arg2 Guest physical address to map the area to.

arg3 Size of the area to map.

Returns:

Upon success this will return P4_E_OK.

In case of error the following error codes can be returned:

P4HWVIRT_E_HYP_INVALID_GUEST_STATE The guest is already initialised.

P4HWVIRT_E_HYP_IOCTL_INVALID_ARGUMENT The passed argument has not a size corresponding to a P4hwvirt_guest_t.

P4HWVIRT_E_HYP_IOCTL_INVALID_USER_ADDRESS The passed argument is not a valid address in the caller address space (invalid pointer).

P4HWVIRT_E_HYP_MAP_INVALID_ADDRESS The given address and size are not aligned to a page or if an address is not in the range of supported addresses (for example higher then 36bit hardware address on 32bit systems).

P4HWVIRT_E_HYP_MAP_OVERMAP The mapping is overriding part of an already existing mapping.

P4HWVIRT_E_HYP_MAP_INCONSISTENT An inconsistency in the page table was found (invalid pointer or flags in existing mappings).

P4HWVIRT_E_HYP_MAP_GIC_OVERRIDE The given physical area contains one of the hardware GIC registers.

P4HWVIRT_E_HYP_NO_HYPMEM There is not enough hypervisor memory to create the guest.

P4HWVIRT_CMD_GUEST_FORWARD_IRQ

Forward an interrupt.

Description:

Forward an hardware interrupt directly to a guest.

The interrupt must be granted to the calling application for this call to be possible.

Parameters:

arg1 Physical interrupt number to forward

arg2 Virtual interrupt number to generate

arg3 unused

arg4 unused

Returns:

Upon success this will return P4_E_OK.

In case of error the following error codes can be returned:

P4HWVIRT_E_HYP_INVALID_GUEST_STATE The guest is already initialised.

P4HWVIRT_E_HYP_IOCTL_INVALID_ARGUMENT The passed argument has not a size corresponding to a P4hwvirt_guest_t.

P4HWVIRT_E_HYP_IOCTL_INVALID_USER_ADDRESS The passed argument is not a valid address in the caller address space (invalid pointer).

P4HWVIRT_E_HYP_IRQ_NOT_GRANTED The interrupts is not granted to the hypervisor.

P4HWVIRT_E_HYP_IRQ_INVALID_IRQID If any of the following conditions are matched:

- The Host interrupt number to be forwarded to the guest is not in the valid range.
- The Guest interrupt number to be generated is not in the valid range.
- The Guest interrupt number is already a forwarded number.

P4HWVIRT_E_HYP_IRQ_CANNOT_ATTACH The Host interrupt number is not granted.

P4HWVIRT_CMD_GUEST_GEN_IRQ

Generate an interrupt.

Description:

Generate a virtual interrupt to the guest.

Parameters:

arg1 Virtual Interrupt number to generate

arg2 Guest Core number to generate the interrupt on (for per core interrupts).

arg3 unused

arg4 unused

Returns:

Upon success this will return P4_E_OK.

In case of error the following error codes can be returned:

P4HWVIRT_E_HYP_INVALID_GUEST_STATE The guest is already initialized.

P4HWVIRT_E_HYP_IOCTL_INVALID_ARGUMENT The passed argument has not a size corresponding to a P4hwvirt_guest_t.

P4HWVIRT_E_HYP_IOCTL_INVALID_USER_ADDRESS The passed argument is not a valid address in the caller address space (invalid pointer).

P4HWVIRT_E_HYP_INVALID_GUEST_COREID The Guest Core number to generate the interrupt on is higher than the number of Guest mapped core.

P4HWVIRT_E_HYP_IRQ_INVALID_IRQID The Virtual Interrupt number is higher than VGIC_MAX_INTERRUPTS.

P4HWVIRT_CMD_GUEST_GET_FREE

Get amount of free hypervisor memory.

Description:

Retrieve the amount of free hypervisor memory.

Parameters:

arg1 Pointer to a P4_address_t where the amount of free hypervisor memory will be written.

arg2 unused

arg3 unused

arg4 unused

Upon success this will return P4_E_OK.

In case of error the following error codes can be returned:

P4HWVIRT_E_HYP_IOCTL_INVALID_ARGUMENT The passed argument has not a size corresponding to a P4hwvirt_guest_t.

P4HWVIRT_E_HYP_IOCTL_INVALID_USER_ADDRESS The passed argument is not a valid address in the caller address space (invalid pointer).

P4HWVIRT_CMD_GUEST_CORE_RUN

Run a guest core.

Description:

Run a guest core.

This call will block until the guest is making an exception that is not handled by the hypervisor.

The core will be executed in the context of the calling thread which means with its priority and time scheduling.

The exception passed is used to setup execution and to return the exception raised by the guest when execution is stopped.

The context passed is updated with the current context on output and is used as core context if the exception run_flags contains the bit P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT

To just run the core for the first time, the exception type must be set like this:

- exception_type = P4VIRT_EXCEPT_NONE
- run_flags = P4HWVIRT_RUNFLAGS_UPDATE_CONTEXT so the initial core context is taken from the given context table
- context passed in arg2 must be filled with the values of the registers wanted when starting the guest (include the PC to define the guest start address).

Parameters:

arg1 Pointer to a P4hwvirt_exception_t

arg2 Pointer to a table of P4_cpureg_t with size P4HWVIRT_NUM_REGS which will contain the core context and be updated when coming back

The core to run is automatically dedicated from the current hardware core we are called on.

Returns:

Upon success this will return P4_E_OK.

In case of error the following error codes can be returned:

P4HWVIRT_E_HYP_INVALID_GUEST_STATE The guest is already initialised.

P4HWVIRT_E_HYP_IOCTL_INVALID_ARGUMENT The passed argument has not a size corresponding to a P4hwvirt_guest_t.

P4HWVIRT_E_HYP_IOCTL_INVALID_USER_ADDRESS The passed argument is not a valid address in the caller address space (invalid pointer).

P4HWVIRT_E_HYP_INVALID_GUEST_COREID The Guest Core number to generate the interrupt on is higher than the number of Guest mapped core.

P4HWVIRT_E_HYP_CANCELED The current thread was stopped or destroyed.

P4HWVIRT_CMD_GUEST_CORE_BREAK

Break a guest core.

Description:

Stop a guest core execution.

Calling this function will make the p4_dev_call of the given coreid to return with a P4HWVIRT_EXCEPT_BREAK exception.

This is usually used to stop a core execution completely or when debugging to see the current context of the given core.

Parameters:

arg1 Core ID to stop

arg2 unused

arg3 unused

Returns:

Upon success this will return P4_E_OK.

In case of error the following error codes can be returned:

P4_E_CONFIG An invalid core ID was given.

14.4.3 Data Type Definitions

P4hwvirt_guestid_t Guest identifier.

Guest uniq identifier. In practice a guest is identified by its PikeOS partition number and there can be only one guest per partition.

P4hwvirt_guest_t Guest declaration structure.

Structure to declare a guest used during the P4HWVIRT_GUEST_INIT.

Some fields are to be set by the application calling P4HWVIRT_GUEST_INIT and some other will be set by the hypervisor during P4HWVIRT_GUEST_INIT and available for the application in the structure when returning from the call.

P4hwvirt_operation_map_t Structure used for a P4HWVIRT_CMD_GUEST_MAP ioctl.

Parameters to map a region to a guest.

P4hwvirt_operation_fw_irq_t Structure used for a P4HWVIRT_CMD_GUEST_FORWARD_IRQ ioctl.

Parameters to forward and interrupt to a guest

P4hwvirt_operation_gen_irq_t Structure used for a P4HWVIRT_CMD_GUEST_GEN_IRQ ioctl.

Paramters to generate an interrupt to a guest

P4hwvirt_operation_run_t Structure used for a P4HWVIRT_CMD_GUEST_CORE_RUN ioctl.

Input and output parameters to run a guest. The context and except content are used to execute the guest in input and are containing when done the information on the reason why the guest was stopped (exception and context)

14.5 HWVIRT User Host interface (ARM specific)

This section describes the communication API and types specific to ARM architecture required to manage a guest from a user application.

14.5.1 Structure Definitions

14.5.1.1 struct P4hwvirt_guest_arch_s

ARM architecture specific guest parameters.

Synopsis:

```
struct P4hwvirt_guest_arch_s {
    P4_phys_addr_t gic_dist_addr;
    P4_phys_addr_t gic_cpu_addr;
    P4_uint32_t iid;
    P4_uint32_t typer;
    P4_uint64_t counter_freq;
};
```

Structure Element Description:

gic_dist_addr Gic Interrupt controller distributor physical address.

This is the physical address at which the guest will see the DIST part of the GIC interrupt controller.

This value can be set by the manager to 0 to use the real hardware value and it will be filled with the real hardware value during INIT. Otherwise the value of this field is used.

gic_cpu_addr Gic Interrupt controller cpu physical address.

This is the physical address at which the guest will see the CPU part of the GIC interrupt controller.

This value can be set by the manager to 0 to use the real hardware value and it will be filled with the real hardware value during INIT. Otherwise the value of this field is used.

iid Guest GIC DIST IID register value.

Value to be returned to the guest when reading the GIC DIST IID register.

typer Guest GIC DIST TYPER register value.

Value to be returned to the guest when reading the GIC DIST TYPER register. In this value the number of guest interrupts will be updated to fit the required number of interrupts for the guest.

counter_freq Timer frequency.

This value is filled during INIT by the hypervisor.

Associated Data Type

P4hwvirt_guest_arch_t ARM architecture specific guest parameters.

14.5.2 Defines

P4HWVIRT_GUEST_FEATURE_ARM_FPU

ARM Guest FPU feature flags.

Description:

When this flag is set in the guest features, the guest can access and use the FPU.

If a guest does not need the FPU, this flag should not be set to save some context save/restore time.

P4HWVIRT_GUEST_FEATURE_ARM_PERF

ARM Guest Performance counter feature flags.

Description:

When this flag is set in the guest features, the guest can access and use the processor performance counters.

Performance counter access could lead to a security leak as the guest could get indirect information on the rest of the system. This should only be activated for debugging purpose.

P4HWVIRT_GUEST_FEATURE_ARM_TRACE

ARM Guest trace feature flags.

Description:

When this flag is set in the guest features, the guest can access and use the processor tracing capabilities.

Trace access could lead to a security leak as the guest could get indirect information on the rest of the system. This should only be activated for debugging purpose.

P4HWVIRT_GUEST_FEATURE_ARM_DEBUG

ARM Guest debug feature flags.

Description:

When this flag is set in the guest features, the guest can access and use the processor debug capabilities.

Debug access could lead to a security leak as the guest could get indirect information on the rest of the system. This should only be activated for debugging purpose.

14.5.3 Data Type Definitions

P4hwvirt_guest_arch_t ARM architecture specific guest parameters.

14.6 HWVIRT KDEV Host interface

This section describes the types, functions and definitions available from an other KDEV driver. Those are used to implement external exception handlers or by IOMMU drivers.

Those functions and definitions are only reacheable from a kernel driver or a PSP.

14.6.1 Function Type Definitions

14.6.1.1 p4hwvirt_exception_handler_init_t

HWVIRT External exception handler init function type.

Synopsis:

```
typedef P4_e_t(* p4hwvirt_exception_handler_init_t)(P4hwvirt_guestid_t guestid,
                                                    P4_cpuid_t numcores,
                                                    P4_address_t global_priv,
                                                    P4_address_t *guest_priv)
```

Description:

This callback is called during the init phase of a guest and before starting the guest.

Usage environment: >=init_prov, <init_complete

This function must NOT be called during the init_complete call of a kdev.

Parameters:

guestid Guest identifier

numcores The number of cores the guest will run on

global_priv Global private data given to register

guest_priv A pointer to a per guest private data that will be passed to other callback during the guest runtime

The function shall return:

P4_E_OK Init successfull

P4_E_MISMATCH The guest is not supported, the handler will not be called again for this guest (init will be called again on next init for the same guest).

Other Error during init, the error will be pushed to the manager and the guest will not start.

14.6.1.2 p4hwvirt_exception_handler_handle_t

HWVIRT External exception handler exception handling function type.

Synopsis:

```
typedef P4_e_t(* p4hwvirt_exception_handler_handle_t)(P4hwvirt_guestid_t guestid,
                                                      P4_cpuid_t coreid,
                                                      P4hwvirt_exception_t *except,
                                                      P4_cpureg_t *context,
                                                      P4_address_t global_priv,
                                                      P4_address_t guest_priv)
```

Description:

This callback is called during runtime if the exception type registered for occured during the guest runtime.

Parameters:

guestid Guest identifier that raised the exception

coreid Core identifier on which the exception occured

except Exception information. This structure is also used used to provide the next action to be done (like the read value for an IO emulation or an other option type when the exception must be forwarded to the manager)

context Guest execution context when the exception occured. It can be modified by the exception handler if needed

global_priv Global private data given to register

guest_priv Private value set during init

The function shall return:

P4_E_OK exception handled, guest execution can continue

P4_E_MISMATCH handler cannot handle the exception, pass over to next handler

P4_E_TRUNC pass over execution to manager Here exception type can be modified to do a special action (core start/stop, poweroff etc)

14.6.1.3 p4hwvirt_exception_handler_exit_t

HWVIRT External exception handler exit function type.

Synopsis:

```
typedef void(* p4hwvirt_exception_handler_exit_t)(P4hwvirt_guestid_t guestid,
                                                  P4_address_t global_priv,
                                                  P4_address_t guest_priv)
```

Description:

This callback is called when a guest is stopped.

Parameters:

guestid Guest identifier

global_priv Global private data given to register

guest_priv Private value set during init

14.6.2 Functions

14.6.2.1 p4hwvirt_register_exception_handler

Register an HWVIRT external exception handler.

Synopsis:

```
p4hwvirt_e_hyp_t p4hwvirt_register_exception_handler(unsigned long except_id,
                                                     p4hwvirt_exception_handler_init_t init,
                                                     p4hwvirt_exception_handler_handle_t handle,
                                                     p4hwvirt_exception_handler_exit_t exit,
                                                     P4_address_t global_priv)
```

Parameters:

except_id Exception ID to register this handler for. This must be one of the existing P4VIRT_EXCEPT_*.

init Guest init callback, this can be NULL

handle Handler callback, this is mandatory

exit Exit callback

global_priv Global private data passed back to all handlers.

Description:

This function should be called once during system init to register an exception handler for a specific exception.

Several exception handlers for the same exception ID can be registered but the handler structure must not be reused.

The exception ID is to be taken out of the P4VIRT_EXCEPT_* defines.

This function is using drv_malloc and must only be called when this service is available.It must be called after P4_BOOT_STAGE_INIT_PROV and before P4_BOOT_STAGE_COMPLETED, if it is used before or within P4_BOOT_STAGE_INIT_PROV it will return P4HWVIRT_E_HYP_INIT_STATE as we can't control the order of kernel drivers init_prov functions execution.

Returns:

Upon success, a call to this function return P4_E_OK. The following error codes can be returned on error:

P4HWVIRT_E_HYP_INIT_STATE if hwvirt is not available

P4HWVIRT_E_HYP_INVAL_EXCEPT if an invalid exception ID is passed.

P4HWVIRT_E_HYP_INVAL_NULL if handle is NULL

14.6.2.2 p4hwvirt_guest_gen_irq

Generate a virtual interrupt to a guest.

Synopsis:

```
p4hwvirt_e_hyp_t p4hwvirt_guest_gen_irq(P4hwvirt_guestid_t guestid,
                                        P4_intid_t intid,
                                        P4_cpuid_t coreid)
```

Parameters:

guestid Guest identifier.

intid Interrupt number to generate.

coreid Core on which the interrupt should be generated. This parameter is only used for interrupt numbers under 32.

Description:

This function will generate a virtual hardware interrupt to the given *guestid*.

Returns:

Upon success, a call to this function return P4_E_OK. The following error codes can be returned on error:

P4HWVIRT_E_HYP_INIT_STATE if hardware virtualization is not available on the system.

P4HWVIRT_E_HYP_INVAL_GUESTID if the given guestid is not valid

P4HWVIRT_E_HYP_INVALID_GUEST_STATE if the given guest is not currently running (no interrupt can be generated)

P4HWVIRT_E_HYP_INVALID_GUEST_COREID if the coreid given is not valid

P4HWVIRT_E_HYP_IRQ_INVALID_IRQID if the requested interrupt number is not valid.

14.6.2.3 p4hwvirt_guest_forward_irq

Forward a physical interrupt to a guest.

Synopsis:

```
p4hwvirt_e_hyp_t p4hwvirt_guest_forward_irq(P4hwvirt_guestid_t guestid,
                                            P4_intid_t phys_irq,
                                            P4_intid_t virt_irq)
```

Parameters:

guestid Guest identifier.

phys_irq Physical interrupt number to forward

virt_irq Virtual interrupt number to generate on the guest

Description:

Calling this will make all interrupt *phys_irq* to generate an interrupt *virt_irq* on the given guest.

This function must be called on each guest init as interrupts forwarded are removed when a guest is stopped.

Returns:

Upon success, a call to this function return P4_E_OK. The following error codes can be returned on error:

P4HWVIRT_E_HYP_INIT_STATE if hardware virtualization is not available on the system.

P4HWVIRT_E_HYP_INVAL_GUESTID if the given guestid is not valid

P4HWVIRT_E_HYP_INVALID_GUEST_STATE if the given guest is not currently initialized

P4HWVIRT_E_HYP_IRQ_INVALID_IRQID if the requested interrupt number is not valid.

P4HWVIRT_E_HYP_IRQ_CANNOT_ATTACH if it is not possible to attach to the given interrupt (someone else is already attached).

14.6.2.4 p4hwvirt_guest_alloc

Allocate per guest memory.

Synopsis:

```
p4hwvirt_e_hyp_t p4hwvirt_guest_alloc(P4hwvirt_guestid_t guestid,
                                      P4_size_t size,
                                      P4_address_t align,
                                      P4_address_t *addr)
```

Parameters:

guestid Guest identifier.

size Size of memory to allocate.

align Alignment of memory to allocate or 0 if no specific alignment is required.

addr Kernel virtual address of the allocated memory.

Description:

This function will allocate some memory from a guest pool. The memory will be lost when the guest is stopped so it must be reallocated on each guest start (usually during init).

Returns:

Upon success, a call to this function return P4_E_OK. The following error codes can be returned on error:

P4HWVIRT_E_HYP_INIT_STATE if hardware virtualization is not available on the system.

P4HWVIRT_E_HYP_INVAL_GUESTID if the given guestid is not valid

P4HWVIRT_E_HYP_INVALID_GUEST_STATE if the given guest is not currently initialized

P4HWVIRT_E_HYP_NO_HYPMEM if there is not enough memory available to do the allocation

14.6.2.5 p4hwvirt_guest_map

Map some memory or IO to a guest.

Synopsis:

```
p4hwvirt_e_hyp_t p4hwvirt_guest_map(P4hwvirt_guestid_t guestid,
                                    P4_phys_addr_t srcaddr,
                                    P4_phys_addr_t dstaddr,
                                    P4_size_t size,
                                    P4_access_t access)
```

Parameters:

guestid Guest identifier

srcaddr Source physical address to map (PA). This must be page aligned.

dstaddr Destination address to map to (IPA). This must be page aligned.

size Size to map. This must be page aligned.

access Cache and access attributes.

Description:

This function will map the given area to be accessible to the guest at the given physical address. The area will be mapped with the access and cache attributes as provided in the access. The guest can define less rights then the one done but not more, for example a RW area can be mapped read-only by the guest but no the other way around. This is also true for caching attributes, a cacheable area can still be mapped uncached.

Returns:

Upon success, a call to this function return P4_E_OK. The following error codes can be returned on error:

P4HWVIRT_E_HYP_INIT_STATE if hardware virtualization is not available on the system.

P4HWVIRT_E_HYP_INVAL_GUESTID if the given guestid is not valid

P4HWVIRT_E_HYP_INVALID_GUEST_STATE if the given guest is not currently initialized

P4HWVIRT_E_HYP_MAP_INVALID_ADDRESS if the given address and size are not aligned to a page or if an address is not in the range of supported addresses (for example higher then 36bit hardware address on 32bit systems).

P4HWVIRT_E_HYP_MAP_GIC_OVERRIDE if the given physical area contains one of the hardware GIC registers

P4HWVIRT_E_HYP_MAP_OVERMAP if the mapping is overriding part of an already existing mapping.

P4HWVIRT_E_HYP_MAP_INCONSISTENT if an inconsistency in the page table was found (invalid pointer or flags in existing mappings).

P4HWVIRT_E_HYP_NO_HYPMEM if there was not enough memory available to allocate the page tables required for this mapping.

14.6.2.6 p4hwvirt_guest_core_wake

Wake up a sleeping guest core or force a context save/restore when running.

Synopsis:

```
p4hwvirt_e_hyp_t p4hwvirt_guest_core_wake(P4hwvirt_guestid_t guestid,
                                          P4_cpuid_t coreid)
```

Parameters:

guestid Guest identifier

coreid Core number to wake up if sleeping

Description:

This function can be used to wake up a guest secondary core sleeping. If the given core is currently running, it will be interrupted and will do a context save/restore (also calling a preemption point.

This can be used to force a rescheduling or to wake up a core from a driver.

Returns:

Upon success, a call to this function return P4_E_OK. The following error codes can be returned on error:

P4HWVIRT_E_HYP_INIT_STATE if hardware virtualization is not available on the system.

P4HWVIRT_E_HYP_INVAL_GUESTID if the given guestid is not valid

P4HWVIRT_E_HYP_INVALID_GUEST_STATE if the given guest is not currently initialized

P4HWVIRT_E_HYP_INVALID_GUEST_COREID if the requested core id is not valid.

14.6.2.7 p4hwvirt_get_guest_ttbr

Retrieve the hardware virtualization guest translation table virtual address for the given client UID guests.

Synopsis:

```
static P4_address_t p4hwvirt_get_guest_ttbr(P4_uid_t client_uid)
```

Parameters:

client_uid [IN] Client UID

Description:

This function will give back a pointer address to a guest translation table. This is to be used by an IOMMU driver that would want to program the same IOMMU mapping then the one used by a specific guest.

This function must be called after a guest initialisation.

Returns:

Upon success, a call to this function return a valid pointer to the guest page table (virtual address).

The following error codes can be returned on error:

- **0** if there is no hardware virtualized guest for the given uid.
- **0** if hardware virtualization is not available on the system
- **0** if the given guest is not initialized

14.6.2.8 p4hwvirt_guest_client_uid

Get the client UID for a guest.

Synopsis:

```
P4_uid_t p4hwvirt_guest_client_uid(P4hwvirt_guestid_t guestid)
```

Parameters:

guestid Guest ID

Description:

This can be used to retrieve the guest client UID to identify the manager task or partition ID.

Parameters:

guestid Guest ID for which the client UID is requested

Returns:

Upon success, the UID of the manager of the guest is returned

P4_UID_INVAL if the guestid is invalid

P4_UID_INVALID if hardware virtualization is not available on the system.

14.6.2.9 p4hwvirt_guest_name

Get the name of a guest.

Synopsis:

```
const char* p4hwvirt_guest_name(P4hwvirt_guestid_t guestid)
```

Parameters:

guestid Guest ID

Description:

This can be used to retrieve the guest name to have a common identifier function for configuration (for properties for example).

Returns:

Upon success, the name of the guest is returned

NULL if the guestid is invalid

NULL if there is not hardware virtual.ization available on the system.

14.6.2.10 p4hwvirt_hyp_strerror

Get a string version of an hwvirt hypervisor error code.

Synopsis:

```
const char* p4hwvirt_hyp_strerror(p4hwvirt_e_hyp_t err)
```

Parameters:

err HWVIRT error code

Description:

This can be used for debugging or to have a message for health monitoring instead of a numerical error code.

Returns:

The error string corresponding to the code or a string saying invalid error code (never returns NULL).

14.7 PSP Guest Interface

This section describes the functions and types to be used by a guest PSP for:

- Communication with the host over the VMM Bus
- Communication with the host over the P4Bus
- Use a PSP console over the VMM Bus or the P4Bus

Those functions are only available from a PSP when the *HWVIRT_GUEST* option is selected.

14.7.1 Functions

14.7.1.1 p4bus_psp_version

Retrieve the P4Bus version from the host.

Synopsis:

```
P4_uint32_t p4bus_psp_version(P4_uint32_t *num_devices)
```

Parameters:

num_devices Number of devices available on P4Bus.

Description:

This function will call the host to retrieve the P4Bus version.

The result should be checked to be compatible with P4BUS_API_VERSION.

Upon success the number of available devices on the bus is returned.

Returns:

Upon success, a call to this function returns the version of the P4Bus.

The following error codes can be returned:

0 if the P4Bus or the VMM bus is not available.

14.7.1.2 p4bus_psp_devinfo

Retrieve the P4Bus device information from the host.

Synopsis:

```
P4_e_t p4bus_psp_devinfo(P4_uint32_t devid,
                         P4_phys_addr_t paddr)
```

Parameters:

devid Device ID

paddr Physical address where to put the device information

Description:

This function will call the host to get the description of the given P4Bus device ID and put in the memory area given in argument.

The given area must have enough space to store the p4bus_device_info_t.

Returns:

Upon success, a call to this function returns P4_E_OK.

The following error codes can be returned:

P4_E_INVAL if the device ID does not exist.

14.7.1.3 p4bus_psp_create_ioring

Create a P4Bus IORING.

Synopsis:

```
P4_e_t p4bus_psp_create_ioring(P4_phys_addr_t paddr,
                               P4_uint32_t *ioring_id)
```

Parameters:

paddr Physical address where to create the IORING

ioring_id ID of the IORING created, to be used to signal the IORING.

Description:

This function will call the host to create a P4Bus IORING at the given physical address.

This IORING can later be used to exchange P4Bus messages.

The given area must have enough space to store the p4bus_ioring_t.

Returns:

Upon success, a call to this function returns P4_E_OK.

The following error codes can be returned:

P4_E_INVAL if there is no more free iorings.

14.7.1.4 p4bus_psp_destroy_ioring

Destroy a P4Bus IORING.

Synopsis:

```
P4_e_t p4bus_psp_destroy_ioring(P4_uint32_t ioring_id)
```

Parameters:

ioring_id ID of the IORING to destroy.

Description:

This function will call the host to destroy a P4Bus IORING.

All messages on the IORING will be lost.

Returns:

Upon success, a call to this function returns P4_E_OK.

The following error codes can be returned:

P4_E_INVAL if the IORING ID is invalid.

14.7.1.5 p4bus_psp_signal_ioring

Signal a P4Bus IORING.

Synopsis:

```
P4_e_t p4bus_psp_signal_ioring(P4_uint32_t ioring_id)
```

Parameters:

ioring_id ID of the IORING to signal.

Description:

This function will call the host to signal a P4Bus IORING.

This function should be called after putting some messages on the IORING to signal the host to process them.

Returns:

Upon success, a call to this function returns P4_E_OK.

The following error codes can be returned:

P4_E_INVAL if the IORING ID is invalid.

14.7.1.6 p4bus_psp_execute_operation

Execute a single P4Bus operation.

Synopsis:

```
P4_e_t p4bus_psp_execute_operation(P4_phys_addr_t operation)
```

Parameters:

operation Physical address containing the description of the operation (type p4bus_operation_t)

Description:

Ask the host to execute a single P4Bus operation.

This should be used when an IORING is not needed.

Returns:

Upon success, a call to this function returns P4_E_OK.

The following error codes can be returned:

P4_E_INVAL if the operation was not possible to execute.

14.7.1.7 vmm_console_psp_init

Initialize console over VMM bus or P4Bus.

Synopsis:

```
P4_e_t vmm_console_psp_init(void)
```

Description:

This function will initialize a PikeOS PSP console working over the VMM Bus or the P4Bus depending on the console number.

Returns:

Upon success, a call to this function returns P4_E_OK.

The following error codes can be returned on error:

P4_E_PAGEFAULT if there was an error converting the internal buffer address to a physical address.

P4_E_CONFIG if the underlying host has a different VMM bus or P4Bus version then us.

P4_E_INVAL if the underlying host has no VMM console available.

P4_E_SYS if there was an error during the P4Bus device initialisation.

P4_E_NOENT if there is no P4Bus device with the requested console ID.

14.7.1.8 vmm_psp_version

Retrieve the VMM protocol version.

Synopsis:

```
P4_uint32_t vmm_psp_version(void)
```

Description:

This function will call the host to retrieve the actual version of the VMM protocol. This shall be compared against VMM_API_VERSION to make sure the host is compatible.

Returns:

Upon success, a call to this function returns a version number.

In case of error (no host underneath for example), a 0 is returned.

14.7.1.9 vmm_psp_system

Retrieve system information from the host.

Synopsis:

```
P4_bool_t vmm_psp_system(P4_cpureg_t *memsize,
                         P4_cpureg_t *numcore,
                         P4_cpureg_t *timer_freq)
```

Parameters:

memsize Available memory size

numcore Number of cores available

timer_freq Timer frequency

Description:

This function will call the host to retrieve some system information and put them in the provided pointers.

Returns:

Upon success, a call to this function returns TRUE.

The following error codes can be returned:

FALSE if one of the pointer is NULL or if there was a communication error with the host.

14.7.1.10 vmm_send_message

Send a message to the host on the VMM bus.

Synopsis:

```
static P4_cpureg_t vmm_send_message(P4_uint32_t id,
                                    P4_cpureg_t *msg)
```

Parameters:

id [IN] Message ID

msg [IN/OUT] Message content and answer

Description:

This function will send the given message ID to the host and will provide back the host answer.

Returns:

The return code from the host is given back. For more information check the VMM protocol.

14.7.1.11 vmm_debug

Send a VMM debug message to the host.

Synopsis:

```
static void vmm_debug(P4_cpureg_t arg0,
                      P4_cpureg_t arg1,
                      P4_cpureg_t arg2)
```

Description:

This function will send a debug message to the host with the provided arguments. This can be used to debug execution as the host will simply output a message on the console and guest execution will continue.

Returns:

Nothing.

14.8 VMM Bus Protocol

This section describes the types and definitions required by the VMM Communication Bus.

The VMM Protocol is based on the hypervisor call and is using 4 registers to pass information:

- register 0 is used to pass a device ID from guest to host and is modified with a return code from host to guest.
- registers 1 to 3 are used to pass a message from guest to host and then a response from host to guest.

Each device is defining its own protocol for the content of the message.

Those definitions are OS generic.

14.8.1 Defines

VMM_API_VERSION
Current VMM API Version.

VMM_API_VERSION_MAJOR_GET (A)
Retrieve the major version of an API version.

VMM_API_VERSION_MINOR_GET (A)
Retrieve the minor version of an API version.

VMM_HVC_NUMBER
HVC number to use for VMM calls.

VMM_HVC_DEBUG
HVC number to use for Debug calls.

VMM_MESSAGE_SIZE
VMM message size.

VMM_DATA_SIZE
Data size in a VMM message.

14.8.2 Data Type Definitions

vmm_dev_id_t Available VMM device ID.

When using the VMM bus, the register 0 is used to provide a device ID to communicate with. This is defining the devices available.

vmm_info_op_t VMM Info device operations.

This defines the available operations on the VMM Info device.

vmm_vmapi_op_t VMAPI device operations.

This gives the available operations on the VMAPI device.

vmm_watchdog_op_t VMM Watchdog device operations.

This defines the available operations on the VMM Watchdog device.

14.8.3 Enumerations

Enumeration type *vmm_dev_id_e*

Available VMM device ID.

When using the VMM bus, the register 0 is used to provide a device ID to communicate with. This is defining the devices available.

Name	Description
VMM_DEV_INFO	This device is used to retrieve system information by the guest. This includes the following information: • Version of the VMM Bus. • Size of memory. • Number of CPUs. See vmm_info_op_t for more information on the protocol.
VMM_DEV_EARLYCON	This device can be used to have a simple early console on a guest that will print on the PikeOS host console. The console is working using an exchange buffer that must be first set to be able to print characters. To use the console the user must: • configure an exchange buffer (to be done once) • write characters in the exchange buffer • send a message with the number of characters to be printed. This is a blocking VMM call handling prints directly and as such should not be used after init of a guest. **Parameters:** **reg[0]** VMM_DEV_EARLYCON **reg[1]** Number of characters to be printed or 0 to setup the console exchange buffer **reg[2]** Physical address of the exchange buffer when reg[1] is 0. **reg[3]** unused **Returns:** **reg[0]** is 0 when no error occured. **reg[0]** is 1 when exchange buffer address is invalid. **reg[0]** is 2 when trying to print without a configured exchange buffer **reg[1-3]** unused
VMM_DEV_P4BUS	This device can be used to communicate with the P4Bus. Check the P4Bus protocol for more information.

VMM_DEV_VMAPI	This device can be used to do some PikeOS vmapi system calls like: • start/stop a partition • stop/reboot the target • change the scheduling schem Those calls are actually made by the manager and depend on the abilities it has. See vmm_vmapi_op_t for more information on the protocol.
VMM_DEV_WATCHDOG	This device is used to communicate with the manager watchdog. See vmm_watchdog_op_t for more information on the protocol.

Enumeration type *vmm_info_op_e*

VMM Info device operations.

This defines the available operations on the VMM Info device.

Name	Description
VMM_INFO_VERSION	Retrieve the VMM API version and the PikeOS version from the host. **Parameters:** **reg[0]** VMM_DEV_INFO **reg[1]** VMM_INFO_VERSION **reg[2]** not used **reg[3]** not used **Returns:** **reg[0]** 0 **reg[1]** VMM_API_VERSION **reg[2]** P4_API_VERSION **reg[3]** not used
VMM_INFO_SYSTEM	Retrieve some system information from the host. **Parameters:** **reg[0]** VMM_DEV_INFO **reg[1]** VMM_INFO_SYSTEM **reg[2]** not used **reg[3]** not used **Returns:** **reg[0]** 0 **reg[1]** Memory size in bytes **reg[2]** Number of cores **reg[3]** Timer clock frequency

Enumeration type *vmm_vmapi_op_e*

VMAPI device operations.

This gives the available operations on the VMAPI device.

Name	Description
VMM_VMAPI_REBOOT_PARTITION	Reboot a PikeOS partition. This is calling vm_reboot. **Parameters:** **reg[0]** VMM_DEV_VMAPI **reg[1]** VMM_VMAPI_REBOOT_PARTITION **reg[2]** Partition ID to reboot **reg[3]** unused **Returns:** **reg[0]** is 0 if no error occured otherwise the PikeOS error code of the call **reg[1-3]** unused
VMM_VMAPI_REBOOT_TARGET	Reboot the complete target. This is calling vm_target_reset. **Parameters:** **reg[0]** VMM_DEV_VMAPI **reg[1]** VMM_VMAPI_REBOOT_TARGET **reg[2-3]** unused **Returns:** **reg[0]** is 0 if no error occured otherwise the PikeOS error code of the call **reg[1-3]** unused
VMM_VMAPI_STOP_PARTITION	Stop a PikeOS partition. This is calling vm_shutdown. **Parameters:** **reg[0]** VMM_DEV_VMAPI **reg[1]** VMM_VMAPI_STOP_PARTITION **reg[2]** Partition ID to stop **reg[3]** unused **Returns:** **reg[0]** is 0 if no error occured otherwise the PikeOS error code of the call **reg[1-3]** unused
VMM_VMAPI_STOP_TARGET	Stop the complete target. This is calling vm_target_reset. **Parameters:** **reg[0]** VMM_DEV_VMAPI **reg[1]** VMM_VMAPI_STOP_TARGET **reg[2-3]** unused **Returns:** **reg[0]** is 0 if no error occured otherwise the PikeOS error code of the call **reg[1-3]** unused

VMM_VMAPI_SET_TIME_SCHED	Change the PikeOS scheduling scheme. This is calling vm_tsched_lookup to find the given scheduling scheme and vm_tsched_change to set it. **Parameters:** **reg[0]** VMM_DEV_VMAPI **reg[1]** VMM_VMAPI_SET_TIME_SCHED **reg[2]** physical address of the string where is set the name of the scheduling scheme to switch to **reg[3]** unused **Returns:** **reg[0]** is 0 if no error occured otherwise the PikeOS error code of the call **reg[1-3]** unused

Enumeration type *vmm_watchdog_op_e*

VMM Watchdog device operations.

This defines the available operations on the VMM Watchdog device.

Name	Description
VMM_WDT_START	Start the watchdog countdown. **Parameters:** **reg[0]** VMM_DEV_WATCHDOG **reg[1]** VMM_WDT_START **reg[2-3]** unused **Returns:** **reg[0]** 0 **reg[1-3]** unused
VMM_WDT_STOP	Stop the watchdog countdown. **Parameters:** **reg[0]** VMM_DEV_WATCHDOG **reg[1]** VMM_WDT_STOP **reg[2-3]** unused **Returns:** **reg[0]** 0 **reg[1-3]** unused

VMM_WDT_REFRESH	Refresh the watchdog and reset the timeout. **Parameters:** **reg[0]** VMM_DEV_WATCHDOG **reg[1]** VMM_WDT_REFRESH **reg[2-3]** unused **Returns:** **reg[0]** 0 **reg[1-3]** unused
VMM_WDT_GET_TIMEOUT	Get the watchdog maximum timeout before which the watchdog must be refreshed. The watchdog should be refreshed at an higher frequency then this value. **Parameters:** **reg[0]** VMM_DEV_WATCHDOG **reg[1]** VMM_WDT_GET_TIMEOUT **reg[2-3]** unused **Returns:** **reg[0]** 0 **reg[1]** Watchdog timeout (only least significant bits on 32bit architecture) **reg[2]** On 32bit architecture this contains the most significant bit of the timeout as it is a 64bit value. **reg[3]** unused
VMM_WDT_SET_TIMEOUT	Set the watchdog maximum timeout before which the watchdog must be refreshed. **Parameters:** **reg[0]** VMM_DEV_WATCHDOG **reg[1]** VMM_WDT_GET_TIMEOUT **reg[2]** Watchdog timeout (only least significant bits on 32bit architecture) **reg[3]** On 32bit architecture this contains the most significant bit of the timeout as it is a 64bit value. **Returns:** **reg[0]** 0 **reg[1-3]** unused

14.9 P4Bus Protocol

This section describes the types and definitions required by the P4Bus Communication Bus.

The P4Bus is based on the VMM Bus to provide an higher level communication layer between a guest and other PikeOS objects.

Those definitions are OS generic.

14.9.1 Structure Definitions

14.9.1.1 struct p4bus_operation_str

p4bus operation structure

Synopsis:

```
struct p4bus_operation_str {
    uint32_t status;
    uint32_t retcode;
    uint16_t type;
    uint16_t flags;
    uint16_t filedesc;
    uint16_t devid;
    uint64_t addr;
    uint64_t size;
    uint64_t param1;
    uint64_t param2;
    uint64_t priv_guest;
};
```

Structure Element Description:

status status of the p4bus operation

retcode return code of the operation

type operation ID

flags operation flags

filedesc file descriptor used in a file operation

devid device id

addr Intermediate physical address

size Size of the area pointed by addr

param1 spare parameter 1

param2 spare parameter 2

priv_guest private parameter to be used by the guest

Associated Data Type

p4bus_operation_t p4bus operation structure

14.9.1.2 struct p4bus_device_info_s

p4bus device structure

Synopsis:

```
struct p4bus_device_info_s {
```

```
    char name[P4BUS_NAMELEN];
    char type[P4BUS_NAMELEN];
    uint32_t p4bus_devid;
    uint32_t type_devid;
    uint32_t interrupt;
    uint32_t opmask;
    uint32_t size;
    uint32_t padding;
};
```

Structure Element Description:

name Device name

type Device type

p4bus_devid Device identifier

type_devid Device type identifier

interrupt Device interrupt number

opmask Device operation mask

size Device operation size

padding align

Associated Data Type

p4bus_device_info_t p4bus device structure

14.9.1.3 struct p4bus_ioring_s

IORING structure.

Note:

curr_ready is handled locally by destination as it is not needed by the source part of the IORING.

Synopsis:

```
struct p4bus_ioring_s {
    uint32_t curr_free;
    uint32_t unused1;
    uint32_t curr_done;
    uint32_t unused2;
    p4bus_operation_t operations[P4BUS_IORING_DEPTH];
};
```

Structure Element Description:

curr_free index of the FREE entry in the ioring

unused1 some padding for 64bit systems

curr_done index of the DONE entry in the ioring

unused2 some padding for 64bit systems

operations p4bus operations of the IORING

Associated Data Type

p4bus_ioring_t IORING structure.

14.9.2 Defines

P4BUS_API_VERSION
Current P4Bus API Version.

P4BUS_API_VERSION_MAJOR_GET (A)
Retrieve the major version of an API version.

P4BUS_API_VERSION_MINOR_GET (A)
Retrieve the minor version of an API version.

OP_FLAGS_SLEEP
Sleep operation flag.

Description:

host signal that after handling this operation it went to sleep, guest must signal host if something is added in the IORING

OP_FLAGS_SIGNAL
Generate IRQ operation flag.

Description:

Guest ask the host to raise an irq once the operation has been handled by the host.

OP_FLAGS_POLL
Poll operation flag.

Description:

Guest is polling on the operation result so host does not need to raise an irq once the operation is done.

This is kept for backward compatibility.

P4BUS_MAX_DEVICES
Maximum number of devices handled by one driver.

P4BUS_MAX_FD
Maximum number of file descriptors per device.

P4BUS_MAX_IORING
Maximum number of iorings.

P4BUS_NAMELEN
Length of name and type of device.

IOCTL_IN

Description:
IOCTL macros

IOCTL_OUT

P4BUS_GET_IOCTL_SIZE_IN (size)

P4BUS_GET_IOCTL_SIZE_OUT (size)

P4BUS_SET_IOCTL_SIZE (size_in, size_out)

P4BUS_FLAGS_OPEN_READ

Description:
Open flags

P4BUS_FLAGS_OPEN_WRITE

P4BUS_FLAGS_OPEN_EXEC

P4BUS_FLAGS_OPEN_MAP

P4BUS_FLAGS_OPEN_RW

P4BUS_FLAGS_OPEN_RWX

P4BUS_FLAGS_MMAP_READ

Description:
Mapping flags

P4BUS_FLAGS_MMAP_WRITE

P4BUS_FLAGS_MMAP_EXEC

P4BUS_FLAGS_MMAP_RW

P4BUS_FLAGS_MMAP_RWX

P4BUS_FLAGS_SEEK_SET

Description:
LSEEK OFFSET FLAGS

P4BUS_FLAGS_SEEK_CUR

P4BUS_FLAGS_SEEK_END

MAPPING_GRANULARITY

Description:
MMAPPING GRANULARITY

MAPPING_MAX_MAPPED_BLOCK

MAPPING_GRANULARITY_MASK

P4BUS_IORING_DEPTH
Number of operations in an IORING.

Description:

For performance reasons this must be a power of 2 so that index can be updated in a single operation and retrieve using the MASK.

P4BUS_IORING_OP_MASK
IORING mask.

Description:

Mask used to retrieve IORING indexes

PIKEOS_IOCTL_DATA_SIZE
Maximum data size supported by a PikeOS IOCTL.

Description:

STAT host driver capacity through IOCTL command

PIKEOS_IOCTL_STRUCT_SIZE
Table size for an IOCTL.

Description:

This is used to do a P4BUS_IOCTL_NATIVE in Linux and the ioctl argument must be a table or uint32_t of size 36 (32 uint32 data and 4 uint32 for args)

PIKEOS_IOCTL_CMD_FIELD
Uint32 table entry for the PikeOS CMD in a P4BUS_IOCTL_NATIVE.

PIKEOS_IOCTL_SIZE_IN_FIELD
Uint32 table entry for the PikeOS Input size in a P4BUS_IOCTL_NATIVE.

PIKEOS_IOCTL_SIZE_OUT_FIELD
Uint32 table entry for the PikeOS Output size in a P4BUS_IOCTL_NATIVE.

PIKEOS_IOCTL_RETURN_FIELD
Uint32 table entry for the PikeOS Return code in a P4BUS_IOCTL_NATIVE.

PIKEOS_IOCTL_DATA_FIELD
Uint32 table entry for the PikeOS IOCTL data (in and out) in a P4BUS_IOCTL_NATIVE.

PIKEOS_IOCTL_CMD_MASK
Mask defining the maximum PikeOS IOCTL command value.

PIKEOS_IOCTL_SIZE_MASK
Mask defining the maximum PikeOS IOCTL data size.

STAT_MAX_UINT32_FIELDS
Maximum number of uint32 values in a stat call.

PIKEOS_STAT_STRUCT_SIZE
Data size when doing a PikeOS stat through a Linux IOCTL.

STAT_TYPE_FIELD
uint32 offset containing the type in a stat

STAT_TOTAL_SIZE_FIELD
uint32 offset containing the total size in a stat

STAT_BLK_SIZE_FIELD
uint32 offset containing the block size in a stat

STAT_QPORT_NB_MESSAGES_SRC_FIELD

STAT_QPORT_NB_MESSAGES_DST_FIELD

STAT_QPORT_DIR_FIELD

STAT_SPORT_SRC_REFRESH_PERIOD_MSB

STAT_SPORT_SRC_REFRESH_PERIOD_LSB

STAT_SPORT_DST_REFRESH_PERIOD_MSB

STAT_SPORT_DST_REFRESH_PERIOD_LSB

STAT_SPORT_SRC_LAST_MSG_VALIDITY

STAT_SPORT_DST_LAST_MSG_VALIDITY

STAT_SPORT_DIR_FIELD

STAT_DIR_TX

STAT_DIR_RX

STAT_DIR_RX_TX

14.9.3 Data Type Definitions

p4bus_operation_status_t Enumeration of operation entry status.
p4bus_operation_t p4bus operation structure
p4bus_device_info_t p4bus device structure
p4bus_command_t Enumeration of p4bus commands.

When the guest sends a command there are 4 values in the VMM message:

- devid : VMM_DEV_P4BUS
- arg[0]: msg[0] p4bus command (p4bus_command_t)
- arg[1]: msg[1]
- arg[2]: msg[2]

p4bus_op_id_t Available operation on a p4bus device.

Each operation except PING will an end operation signal. Any non blocking concept depends on non blocking behaviour on PikeOS side but will still generate a signal in linux.

Depending on the other side support, some operations may be unsupported (retcode: P4BUS_E_UNSUPP) or can simply be emulated (for OPEN or CLOSE for example).

p4bus_op_error_t Operation Errors.

Any operation returns an error of the following type.

p4bus_ioring_t IORING structure.

Note:

curr_ready is handled locally by destination as it is not needed by the source part of the IORING.

14.9.4 Enumerations

Enumeration type *p4bus_operation_status_e*

Enumeration of operation entry status.

Name	Description
P4BUS_OPERATION_FREE	operation entry is free
P4BUS_OPERATION_READY	operation entry is ready to be processed
P4BUS_OPERATION_DONE	operation has been processed

Enumeration type *p4bus_command_e*

Enumeration of p4bus commands.

When the guest sends a command there are 4 values in the VMM message:

- devid : VMM_DEV_P4BUS
- arg[0]: msg[0] p4bus command (p4bus_command_t)
- arg[1]: msg[1]
- arg[2]: msg[2]

Name	Description
P4BUS_COMMAND_INFO_VERSION	**Parameters:** **msg[0]** P4BUS_COMMAND_INFO_VERSION **msg[1]** not used **msg[2]** not used **Returns:** **msg[0]** p4bus API Version **msg[1]** Number of devices on the p4bus **msg[2]** unused
P4BUS_COMMAND_INFO_DEVICE	**Parameters:** **msg[0]** P4BUS_COMMAND_INFO_DEVICE **msg[1]** physical address where to store the device info (size of p4bus_device_info_t) **msg[2]** p4bus device ID **Returns:** **msg[0]** not used **msg[1]** physical address where is stored the device info (size of p4bus_device_info_t) **msg[2]** not used

P4BUS_COMMAND_CREATE_IORING	This call will inform the manager to attach an operation thread to the given IORING. The ioring must be initialized by the guest before. **Parameters:** **msg[0]** P4BUS_COMMAND_CREATE_IORING **msg[1]** physicall address of the IORING **msg[2]** not used **Returns:** **msg[0]** ID of the created IORING **msg[1]** not used **msg[2]** not used
P4BUS_COMMAND_DESTROY_IORING	All operations ongoing are stopped on the host (no signaling) **Parameters:** **msg[0]** P4BUS_COMMAND_DESTROY_IORING **msg[1]** IORING ID **msg[2]** not used **Returns:** **msg[0]** not used **msg[1]** not used **msg[2]** not used
P4BUS_COMMAND_SIGNAL_IORING	**Parameters:** **msg[0]** P4BUS_COMMAND_SIGNAL_IORING **msg[1]** IORING ID **msg[2]** not used **Returns:** **msg[0]** not used **msg[1]** not used **msg[2]** not used
P4BUS_COMMAND_EXECUTE_OPERATION	All operations are done using the same thread **Parameters:** **msg[0]** P4BUS_COMMAND_EXECUTE_OPERATION **msg[1]** physical address of the operation (p4bus_operation_t) **msg[2]** not used **Returns:** **msg[0]** operation ID (to be used for cancel) **msg[1]** not used **msg[2]** not used

P4BUS_COMMAND_CANCEL_OPERATION	**Parameters:** **msg[0]** P4BUS_COMMAND_CANCEL_OPERATION **msg[1]** ID of the operation to cancel **msg[2]** not used **Returns:** **msg[0]** not used **msg[1]** not used **msg[2]** not used

Enumeration type *p4bus_op_id_e*

Available operation on a p4bus device.

Each operation except PING will an end operation signal. Any non blocking concept depends on non blocking behaviour on PikeOS side but will still generate a signal in linux.

Depending on the other side support, some operations may be unsupported (retcode: P4BUS_E_UNSUPP) or can simply be emulated (for OPEN or CLOSE for example).

Name	Description
P4BUS_OP_PING	**Parameters:** **param1** unused **param2** unused **addr** unused (must be NULL) **size** unused (must be 0)
P4BUS_OP_OPEN	**Parameters:** **param1** open flags **param2** unused **addr** unused (must be NULL) **size** unused (must be 0)
P4BUS_OP_CLOSE	**Parameters:** **param1** unused **param2** unused **addr** unused (must be NULL) **size** unused (must be 0)
P4BUS_OP_CANCEL	**Parameters:** **param1** operation ID to cancel **param2** unused **addr** unused (must be NULL) **size** unused (must be 0)

P4BUS_OP_READ	**Parameters:** **param1** offset **param2** if not 0, read at offset in param1 **addr** virtual address of buffer **size** buffer size / size read
P4BUS_OP_NB_READ	**Parameters:** **param1** offset **param2** if not 0, read at offset in param1 **addr** virtual address of buffer **size** buffer size / size read
P4BUS_OP_WRITE	**Parameters:** **param1** offset **param2** if not 0, write at offset in param1 **addr** virtual address of buffer **size** buffer size / size written
P4BUS_OP_NB_WRITE	**Parameters:** **param1** offset **param2** if not 0, write at offset in param1 **addr** virtual address of buffer **size** buffer size / size written
P4BUS_OP_IOCTL	**Parameters:** **param1** command **param2** size_in (on the most significant 16 bits) | size_out (on the least significant 16 bits) **addr** data virtual address **size** data size
P4BUS_OP_MMAP	**Parameters:** **param1** offset **param2** flags **addr** unused (must be NULL)/physical address **size** size to map

P4BUS_OP_STAT	**Parameters:** **param1** unused **param2** unused **addr** stat buffer address **size** stat buffer size / size written
P4BUS_OP_LSEEK	**Parameters:** **param1** offset **param2** origin **addr** unused (must be NULL) / new file position **size** unused (must be 0)
P4BUS_OP_NUM	Rest of operation numbers are custom and can be used for special purpose. Operations are used as bit mask values in a 32 bit so max value is 31.

Enumeration type *p4bus_op_error_e*

Operation Errors.

Any operation returns an error of the following type.

Name	Description
P4BUS_E_OK	Operation done correctly
P4BUS_E_UNSUPP	Operation not supported by driver
P4BUS_E_INVAL_ARG	Invalid operation argument
P4BUS_E_BUS	Bus error during transfer
P4BUS_E_BUSY	Operation ongoing
P4BUS_E_TRUNC	Operation truncated
P4BUS_E_P4	PikeOS Ukernel Error P4_e_t = error - P4BUS_E_P4

14.9.5 Functions

14.9.5.1 p4bus_ioring_init

Initialize an ioring.

Synopsis:

```
static void p4bus_ioring_init(p4bus_ioring_t *ioring)
```

Description:

This function must be called the before the P4BUS command P4BUS_COMMAND_CREATE_IORING on the targeted IORING.

Parameters:

***ioring** pointer on the ioring structure to initialize

14.9.5.2 p4bus_ioring_has_free

Check if a free operation is available.

Synopsis:

```
static uint32_t p4bus_ioring_has_free(p4bus_ioring_t *ioring)
```

Description:

Parameters:

***ioring** pointer on the ioring targeted

Returns:

TRUE(0x1) : if a free operation is available

FALSE(0x0) if there is no free operation

14.9.5.3 p4bus_ioring_get_free

Retrieve a free operation.

Synopsis:

```
static p4bus_operation_t* p4bus_ioring_get_free(p4bus_ioring_t *ioring)
```

Description:

This function must be called in a critical section.

Parameters:

***ioring** pointer on the ioring targeted

Returns:

return the address of the first free operation available

NULL : if no free operation is available

14.9.5.4 p4bus_ioring_push_free

Push a free operation in the IORING.

Synopsis:

```
static void p4bus_ioring_push_free(p4bus_operation_t *ioring_entry)
```

Description:

Parameters:

***ioring_entry** pointer on the ioring operation to push as free

14.9.5.5 p4bus_ioring_has_ready

Check if a ready operation is available.

Synopsis:

```
static uint32_t p4bus_ioring_has_ready(p4bus_ioring_t *ioring,
                                       uint32_t *curr_ready)
```

Description:

Parameters:

***ioring** pointer on the ioring targeted

***curr_ready** pointer to the ready counter of the ioring

Returns:

TRUE(0x1) : if a ready operation is available

FALSE(0x0) if there is no ready operation

14.9.5.6 p4bus_ioring_get_ready

Retrieve ready-to-process operation.

Synopsis:

```
static p4bus_operation_t* p4bus_ioring_get_ready(p4bus_ioring_t *ioring,
                                                 uint32_t *curr_ready)
```

Description:

This function must be called in a critical section.

Parameters:

***ioring** pointer on the ioring targeted

***curr_ready** pointer to the ready counter of the ioring

Returns:

return the address of the first ready operation available

NULL : if no ready operation is available

14.9.5.7 p4bus_ioring_push_ready

Push a ready-to-process operation.

Synopsis:

```
static void p4bus_ioring_push_ready(p4bus_operation_t *ioring_entry)
```

Description:

Parameters:

***ioring_entry** pointer on the ioring operation to push as ready

14.9.5.8 p4bus_ioring_has_done

Check if a done operation is available.

Synopsis:

```
static uint32_t p4bus_ioring_has_done(p4bus_ioring_t *ioring)
```

Description:

Parameters:

***ioring** pointer on the ioring targeted

Returns:

TRUE(0x1) : if a done operation is available

FALSE(0x0) if there is no done operation

14.9.5.9 p4bus_ioring_get_done

Retrieve done operation.

Synopsis:

```
static p4bus_operation_t* p4bus_ioring_get_done(p4bus_ioring_t *ioring)
```

Description:

This function must be called in a critical section.

Parameters:

***ioring** pointer on the ioring targeted

Returns:

return the address of the first done operation available

NULL : if no done operation is available

14.9.5.10 p4bus_ioring_push_done

Push a done operation in the IORING.

Synopsis:

```
static void p4bus_ioring_push_done(p4bus_operation_t *ioring_entry)
```

Description:

Parameters:

***ioring_entry** pointer on the ioring operation to push as done