Harman User Guide

Harman Device Virtualization for Connected Vehicles

How to Move a Pass-Through Device from System Domain to User Domain

Software Version 12.1
Doc. Rev. 2.5 / 556

Reference: DV-0007

Date: 3/24/2022

HARMAN

Table of Contents

© 2022 Harman. All Rights Reserved **Confidential**

Table of Figures

© 2022 Harman. All Rights Reserved

Confidential

1 Introduction

1.1 This document

The Harman Device Virtualization for Connected Vehicles product is a high-performance, real-time device virtualization software that targets automotive systems. It enables multiple guest operating systems (Guest OSes) to run simultaneously on a single hardware platform featuring Intel or ARM processors.

In a very basic configuration, one VM and its guest OS, both called *system (SYS) domain*, are controlling most of the hardware devices, while another VM and its guest OS, both called *user (USR) domain*, have no access to hardware devices. But this simplistic picture, which is usually the starting point of any virtualization project, can evolve depending on the requirements, and USR domain can take ownership of hardware devices, thus leading to a design where hardware devices are split between both SYS domain and USR domain. This document describes how to configure the Harman Device Virtualization Runtime Software to move the ownership of a hardware device from SYS domain to USR domain.

Note that this document only exposes a solution to *move* the ownership of a device from one guest OS to another, and does not provide a solution to *share* the device between guest OSes. Since device resources will be directly accessed by its native driver without any sharing support, the device is called a *pass-through* device.

The two figures below illustrate the move of device B from SYS domain to USR domain.

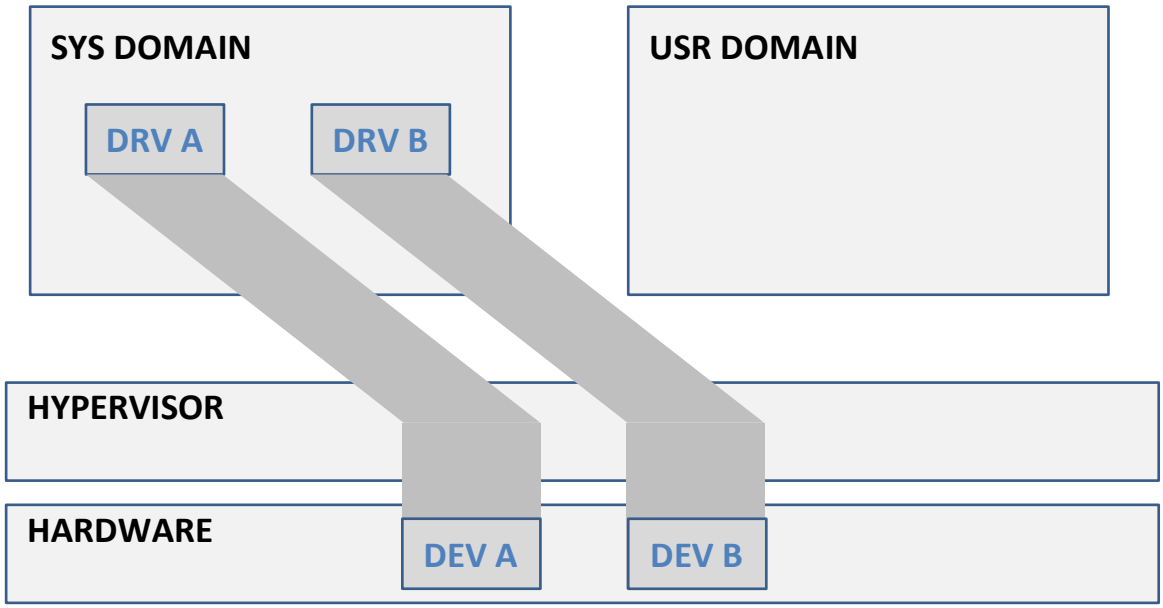


Figure 1-1 - Initial state, before the move of device B from SYS domain to USR domain

© 2022 Harman. All Rights Reserved

Confidential

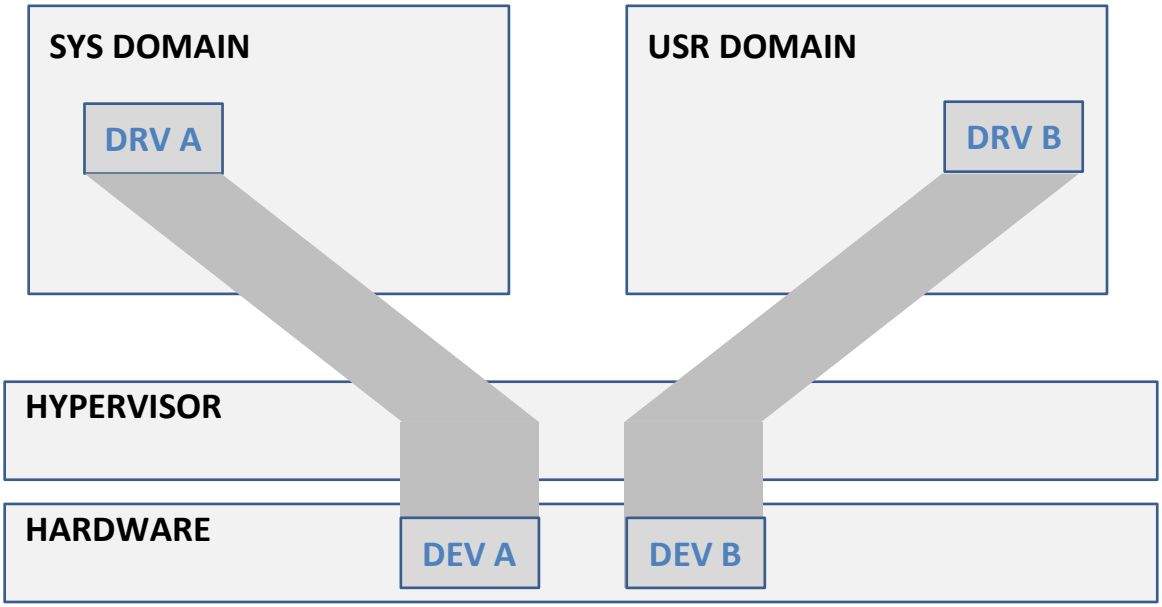


Figure 1-2 - Target state, after the move of device B from SYS domain to USR domain

1.2 Related Documentation

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

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

© 2022 Harman. All Rights Reserved

Confidential

1.3 Support

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

2 Prerequisites

The reader should be able to build and launch two guest OSes on top of Harman Hypervisor: a SYS domain running all its devices as pass-through, and a USR domain.

The reader should also be able to access and modify the following configurations:

- Hypervisor configuration
- SYS domain virtual platform (vplatform) configuration
- USR domain virtual platform (vplatform) configuration
- SYS domain Guest OS device tree
- USR domain Guest OS device tree

3 Overview

This document goes through the different steps that are required to move a pass-through device from SYS domain to USR domain:

- Interrupt Configuration
- IO Memory Configuration
- Device Node Configuration
- Clock Configuration

Often the migrated devices uses some sub-devices. For instance, a serial line device might use a power-domain device because the serial line needs to be powered on. Or it may depend on a baud rate generator (clock), in which case the serial line device driver requires to program the baud rate generator at the proper rate. In addition, these 'sub-devices' might be used by other devices (other serial lines in the above example). If the 'sub-device' is used by devices owned by different VMs then this 'sub-device' shall be virtualized. Hopefully, The "Product" provides virtualized sub-device (like VCLK or VPD), which eases the migration of devices. So most of the work for migrating a device from one VM to the other requires to configure these "sub-device" and this is described in the following chapters.

- Power Domain Configuration

4 Pin ConfigurationInterrupt Configuration

One of the first steps in the process of moving a device from SYS domain to USR domain is to configure the virtual platform to forward device interrupts to USR domain instead of SYS domain.

4.1 Understand Interrupt Mapping

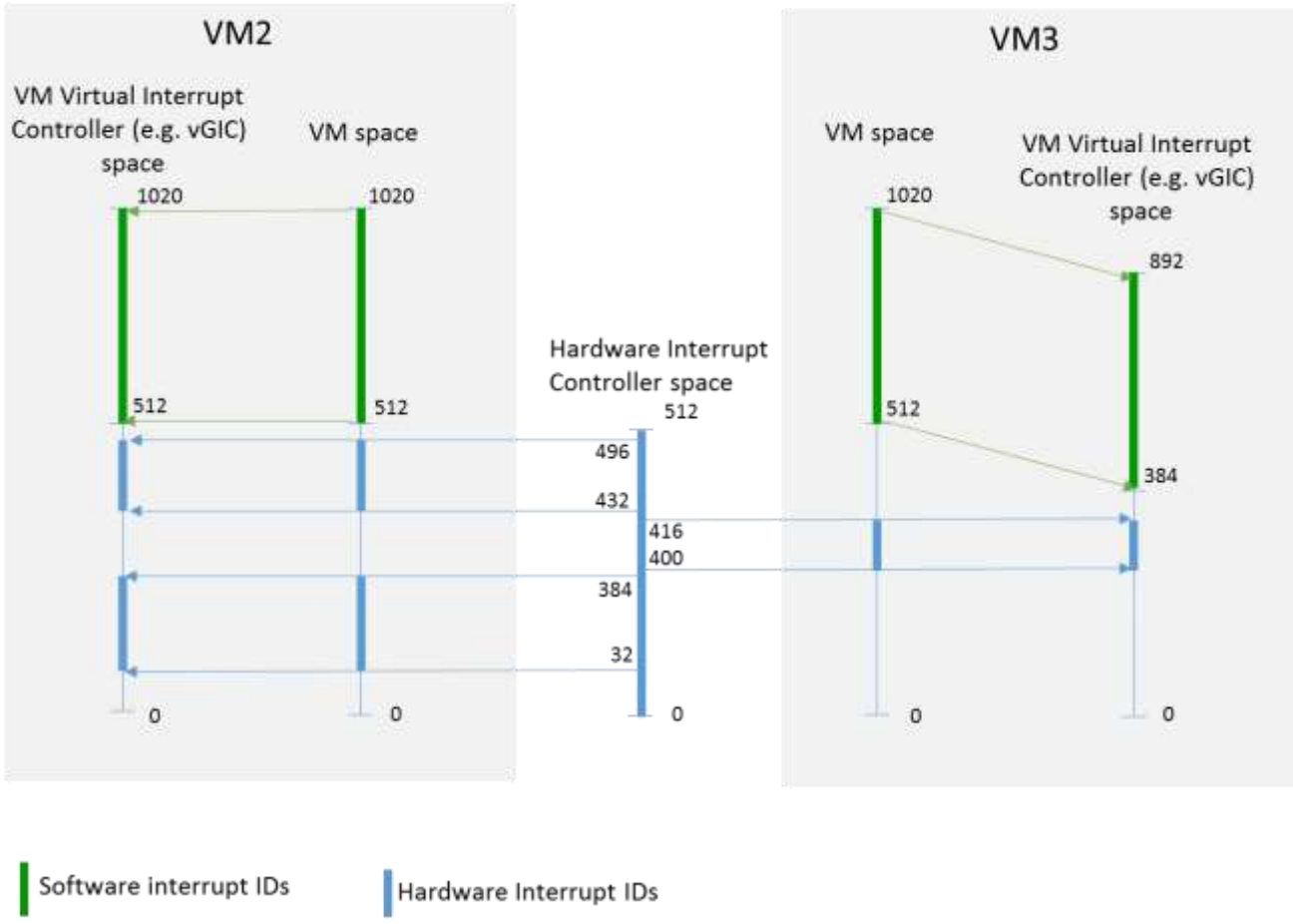
From the Hypervisor point of view, an interrupt is known as a *cross interrupt,* also called *XIRQ*, and is part of the range [0, 1020), which is composed of two disjoint ranges, e.g:

© 2022 Harman. All Rights Reserved

Confidential

- [0, 512[1]): physical interrupts
- [512, 1020[2]): software interrupts, typically used for communication between virtual drivers

Interrupt mapping configuration defines, for each XIRQ, if it is forwarded to a given VM, and under which virtual Interrupt Controller IRQ ID. Below picture provides an example of interrupt mapping configuration of two VMs:



As seen in the above picture, hardware interrupts [0, 512) are partitioned between different guest OSes: one hardware interrupt ID may be assigned only to one VM.

Software interrupts, on the other hand, obey to a different rule: each VM is provided with an independent space of Software Interrupts. Thus the same software interrupt ID may be reused in multiple VMs.

In a large majority of use cases, an identical mapping will be defined by each VM for hardware and software interrupts. One reason of a non-identical mapping might be a limitation of the number of supported interrupt lines in the guest OS kernel. In above picture, VM3 only supports interrupts up to 892, so we use a non-identical mapping.

Interrupt mapping is defined in each VM virtual platform configuration by one or several IRQ mapper device tree nodes, each of those nodes defining the mapping for one range of interrupts.

[1] 512 is the typical default value, but it depends on the board
[2] The maximum interrupt range goes up to 1023, however it can be limited to a smaller range by configuration.

© 2022 Harman. All Rights Reserved

Confidential

An IRQ mapper node always has a `compatible` property set to `vl,interrupts-mapper`, and describes the mapping using three other properties:

- `interrupts`: defines the first GIC IRQ in the interval, as specified by Linux interrupt client binding. On ARM, this is usually defined by three integers: interrupt type (0: SPI, 1: PPI), interrupt number, and flags. Interrupt number has to be specified according to its type:
  - PPIs are ARM interrupts within [16-31], but since PPI type is specified as the first integer of the `interrupts` property, they are numbered within [0-15].
  - SPIs are ARM interrupts within [32-1019], but since SPI type is specified as the first integer of the `interrupts` property, they are numbered within [0-987].
- `vl,interrupts`: defines the first XIRQ in the interval. This property is optional. If omitted, an identical mapping will be created.
- `#vl,interrupts-count`: defines the number of interrupts in the range.

As an example, the mapping shown in the above picture is fully described by the following device tree nodes:

- In VM2 virtual platform configuration:

```
vl,xirq-32-384 {
   compatible = "vl,interrupts-mapper";
   interrupts = <0 0 0>;       // SPI0 = 0 + 32 = IRQ32
   #vl,interrupts-count = <352>;
};

vl,xirq-432-496 {
   compatible = "vl,interrupts-mapper";
   interrupts = <0 400 0>;          // SPI400 = 400 + 32 = IRQ432
   #vl,interrupts-count = <64>;
};

vl,xirq-512-1020 {
   compatible = "vl,interrupts-mapper";
   interrupts = <0 480 0>;          // SPI480 = 480 + 32 = IRQ512
   #vl,interrupts-count = <508>;
};
```

- In VM3 virtual platform configuration:

```
vl,xirq-400-416 {
   compatible = "vl,interrupts-mapper";
   interrupts = <0 368 0>;          // SPI368 = 368 + 32 = IRQ400
   #vl,interrupts-count = <16>;
};

vl,xirq-512-1020 {
   compatible = "vl,interrupts-mapper";
   vl,interrupts = <512>;
```

© 2022 Harman. All Rights Reserved

Confidential

```
    interrupts = <0 352 0>;          // SPI352 = 352 + 32 = IRQ384
    #vl,interrupts-count = <508>;
};
```

4.2 Identify Device Interrupts

For example, let's suppose we plan to move Exynos ADC (Analog To Digital Converter) device from SYS domain to USR domain. We first need to know what interrupts are handled by this device. This information can easily be found in SYS domain device tree that includes the below node:

```
exynos_adc: adc@13620000 {
        compatible = "samsung,exynos-adc-v2";
        reg = <0x0 0x13620000 0x100>;
        interrupts = <0 423 0>;
        #io-channel-cells = <1>;
        io-channel-ranges;
        clocks = <&clock 209>;
        clock-names = "gate_adcif";
};
```

This node shows that Exynos ADC device is using SPI interrupt 423, i.e global IRQ455.

4.3 Configure SYS Domain

By default, SYS domain virtual platform configuration includes one or several nodes that instruct the hypervisor to forward **all** PPI and SPI interrupts to SYS domain. Below is an example of those nodes:

```
// Forward all PPI interrupts
vl,xirq-ppi {
    compatible = "vl,interrupts-mapper";
    interrupts = <1 0 0>;
    #vl,interrupts-count = <16>;
};

// Forward all SPI interrupts
vl,xirq-spi {
    compatible = "vl,interrupts-mapper";
    interrupts = <0 0 0>;

    // There are 988 SPI interrupts within [32-1020].
    #vl,interrupts-count = <988>;
};
```

To remove SPI 423 from the range of forwarded interrupts, we need to split node “vl,xirq-spi” into two nodes defining two ranges skipping SPI 423:

© 2022 Harman. All Rights Reserved

Confidential

```
    // Forward SPI interrupts [0, 423)
    vl,xirq-spi-0-423 {
       compatible = "vl,interrupts-mapper";
       interrupts = <0 0 0>;
       #vl,interrupts-count = <423>;
    };

    // Forward SPI interrupts [424, 988)
    vl,xirq-spi-424-988 {
       compatible = "vl,interrupts-mapper";
       interrupts = <0 424 0>;
       #vl,interrupts-count = <564>;
    };
```

4.4 Configure USR Domain

By default, USR domain vplatform configuration could typically include the following node that only maps the first 256 software interrupts:

```
    vl,xirq-512-768 {
       compatible = "vl,interrupts-mapper";
       interrupts = <0 480 0>;
       #vl,interrupts-count = <256>;
    };
```

To enable the forwarding of SPI 423 to USR domain, we need to add the following node to USR domain vplatform configuration:

```
vl,xirq-spi@423 {
       compatible = "vl,interrupts-mapper";
       interrupts = <0 423 0>;
       #vl,interrupts-count = <1>;
};
```

4.5 The “vl,disabled” property

The previously described procedure allows you to map interrupts to different VMs. However, during development phases, when a lot of devices have to be moved one at a time from SYS domain to USR domain, continuously breaking IRQ mapper nodes into two new nodes to exclude a single interrupt could quickly become a tedious work. This is because you have to recalculate range size and limits for the two new nodes in SYS domain configuration. To solve this problem, you can use the property “vl,disabled” in an IRQ mapper node, that instructs the node to actually ***unmap*** the specified range instead of mapping it.

In our example, instead of breaking SYS domain virtual platform configuration to exclude SPI 423, you can just keep the original nodes that are mapping **all** PPI and SPI interrupts, and add an extra node that unmap a single interrupt:

```
vl,xirq-disable-spi@423 {
       compatible = "vl,interrupts-mapper";
       interrupts = <0 423 0>;
       #vl,interrupts-count = <1>;
```

© 2022 Harman. All Rights Reserved

Confidential

```
        vl,disabled;
};
```

For more information about interrupt configuration, please refer to the section Configuration > VM Virtual Platform bindings > Virtual Platform IRQ Mapper Node in the Hypervisor Reference Manual.

5 IO Memory Configuration

5.1 Understand IO Memory Mapping

The Harman Device Virtualization for Connected Vehicles product is typically based on a 2-stage MMU that translates virtual addresses (VA) to intermediate physical addresses (IPA), which is then translated to physical addresses (PA). An application is typically accessing a VA that has been mapped to an IPA by its guest OS. The guest OS is typically accessing an IPA that has been mapped to a PA by the hypervisor. This mechanism is illustrated by the following figure.

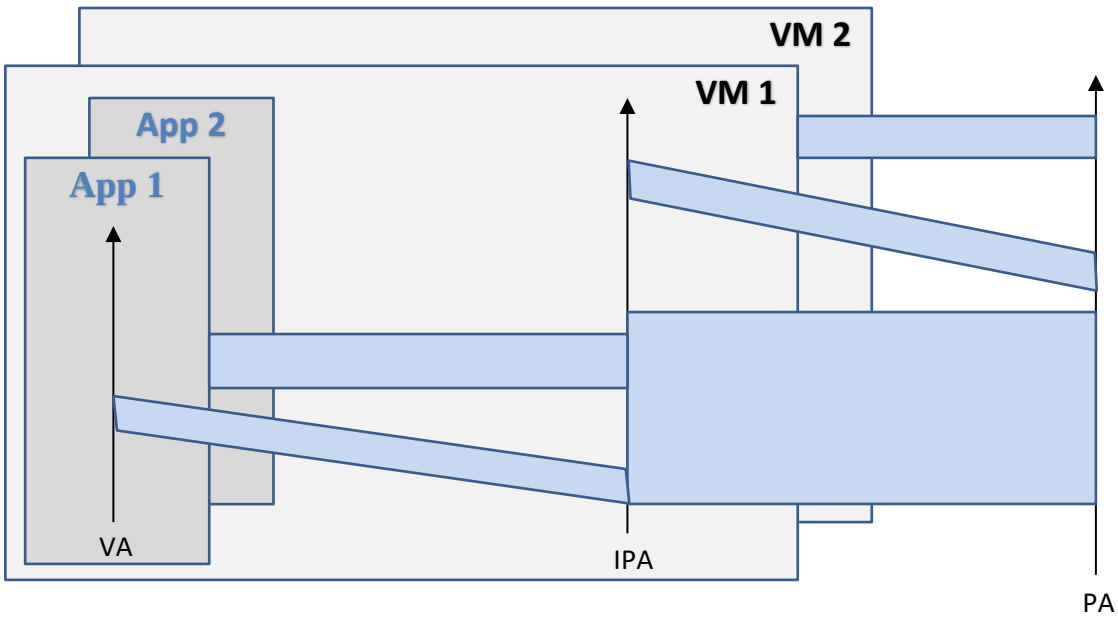


Figure 5-1 Address virtualization using a 2-stage MMU

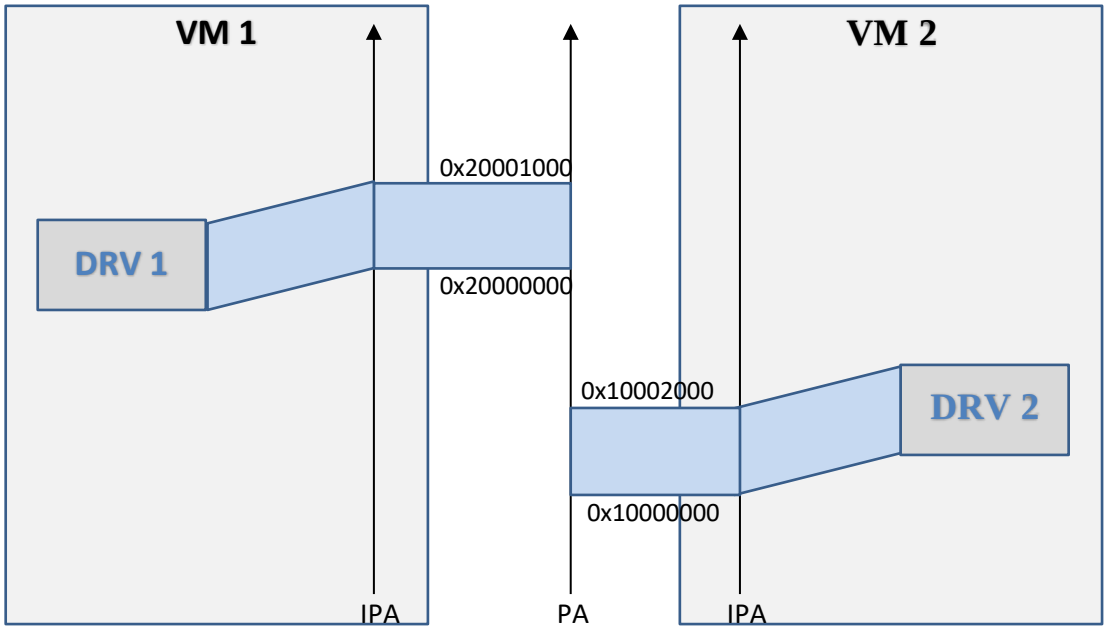


Figure 5-2 - IPA to PA mapping example

IPA to PA mapping is configured in each VM virtual platform configuration. For example, the mapping shown in the above figure will be defined by the following device nodes[3]:

- In VM1 virtual platform configuration:

```
vl,io-memory-vm1 {
        compatible = "vl,io-memory";
        reg = <0x20000000 0x1000>;
};
```

- In VM2 virtual platform configuration:

```
vl,io-memory-vm2 {
        compatible = "vl,io-memory";
        reg = <0x10000000 0x2000>;
};
```

5.2 Identify Device IO Memory

IO memory of the device we want to move from SYS domain to USR domain can easily be identified by its `reg` property in its target configuration node in SYS domain device tree. Below example shows that Exynos ADC IO Memory starting address is 0x13620000, and its size is 0x100 bytes.

```
        exynos_adc: adc@13620000 {
                compatible = "samsung,exynos-adc-v2";
                reg = <0x0 0x13620000 0x100>;
```

[3] Assuming that both #address-cells and #size-cells properties are equal to 1.

© 2022 Harman. All Rights Reserved

Confidential

```
                interrupts = <0 423 0>;
                #io-channel-cells = <1>;
                io-channel-ranges;
                clocks = <&clock 209>;
                clock-names = "gate_adcif";
        };
```

5.3 Configure SYS Domain

Default SYS domain virtual platform configuration will typically include a node that is allowing SYS domain to access the whole IO memory area:

```
vl,io-memory@0-20000000 {
        compatible = "vl,io-memory";
        reg = <0x0 0x20000000>;
};
```

To ensure SYS domain won't be able to access ADC device IO memory registers, we need to add a node to remove the range [0x13620000, 0x13620100[from the allowed addresses:

```
vl,io-memory@customized {
        compatible = "vl,io-memory";
        reg = <0x13620000 0x100>;

        vl,disabled;
};
```

By default the *"vl,io-memory"* compatible regions are page (4K) aligned. For example, the following two node definitions are equivalent:

```
vl,io-memory@customized {
        compatible = "vl,io-memory";
        reg = <0x13620000 0x100>;
        vl,disabled;
};
vl,io-memory@customized {
        compatible = "vl,io-memory";
        reg = <0x13620000 0x1000>;
        vl,disabled;
};
```

As a result the whole page range [0x13620000, 0x13621000[will be removed.

Sometimes you need to disable access to a specific sub-range of registers within a page. An optional *vl,alignment* property may explicitly specify the alignment as a power-2 value which shall be greater or equal to 4.

```
vl,io-memory@10221004 {
        compatible = "vl,io-memory";
        reg = <0x10221004 0x4>;
        vl,alignment = <4>;
        vl,disabled;
};
```

© 2022 Harman. All Rights Reserved

Confidential

For example, the above node disables access to only one register located at the address 0x10221004, and still enables access to all other registers located in the page 0x10221000.

Hypervisor uses the second stage MMU when the configuration enables access to an entire page. On the other hand, when the configuration enables access to a subset of registers only, Hypervisor intercepts[4] any access made by the Guest OS to any register in the page, checks whether the configuration enables access to the register, and eventually either performs the access on behalf of the Guest OS, or aborts it.

5.4 Configure USR Domain

USR domain virtual platform needs to be configured to allow USR Domain to access ADC device IO memory, adding the following node:

```
vl,io-memory@customized {
        compatible = "vl,io-memory";
        reg = <0x13620000 0x1000>;
};
```

In order to enable access a specific sub-range of registers within a page you may use the optional *vl,alignment* property:

```
vl,io-memory@10221004 {
        compatible = "vl,io-memory";
        reg = <0x10221004 0x4>;
        vl,alignment = <4>;
};
```

For example, the above node enables access to only one register located at the address 0x10221004, and still disables access to all other registers located in the page 0x10221000.

For more information about I/O memory configuration, please refer to the section *Configuration > VM Virtual Platform bindings > Virtual Platform I/O Memory Node* in the Hypervisor Reference Manual.

6 Device Node Configuration

Now that interrupt forwarding, and IO memory permissions are correctly configured, you can enable Exynos ADC driver in USR domain Linux system, and eventually move the following device tree node from SYS domain device tree to USR domain device tree:

```
        exynos_adc: adc@13620000 {
                compatible = "samsung,exynos-adc-v2";
                reg = <0x0 0x13620000 0x100>;
                interrupts = <0 423 0>;
                #io-channel-cells = <1>;
                io-channel-ranges;
                clocks = <&clock 209>;
                clock-names = "gate_adcif";
        };
```

[4] It is worth knowing that interception and emulation has a performance impact.

© 2022 Harman. All Rights Reserved

Confidential

Unfortunately, USR domain device tree will not build with the above node, because it depends on the node ***clock*** that is only defined in SYS domain device tree. This problem will be solved in the next section.

7 Clock Configuration

7.1 The Clock Dependency Issue

As described in the previous section, moving a device node from SYS domain to USR domain can typically lead to dependency issues. In our example, Exynos ADC device depends on the clock node ***`clock`*** that is only defined in SYS domain device tree. You can think about a few solutions to solve this problem:

- move the clock node from SYS domain to USR domain: it will typically lead to dependency issues in SYS domain due to other devices that still rely on the missing clock node.
- remove the clock property: might lead to driver crash, or malfunction.
- modify the clock property to point to a newly added fake clock: might lead to driver crash, or malfunction.
- virtualizing the clock using the clock back-end / front-end driver: this is usually the safest solution and will be described in the next section.

7.2 Virtualizing the Clock

Clock virtualization allows USR domain ADC driver to rely on its original clock defined in SYS domain device tree, thus solving the dependency issue previously described.

The next two figures illustrate how clock virtualization works: it is based on a clock front-end driver and a clock back-end driver that allow a device owned by the USR domain to still rely on a clock owned by the SYS domain.

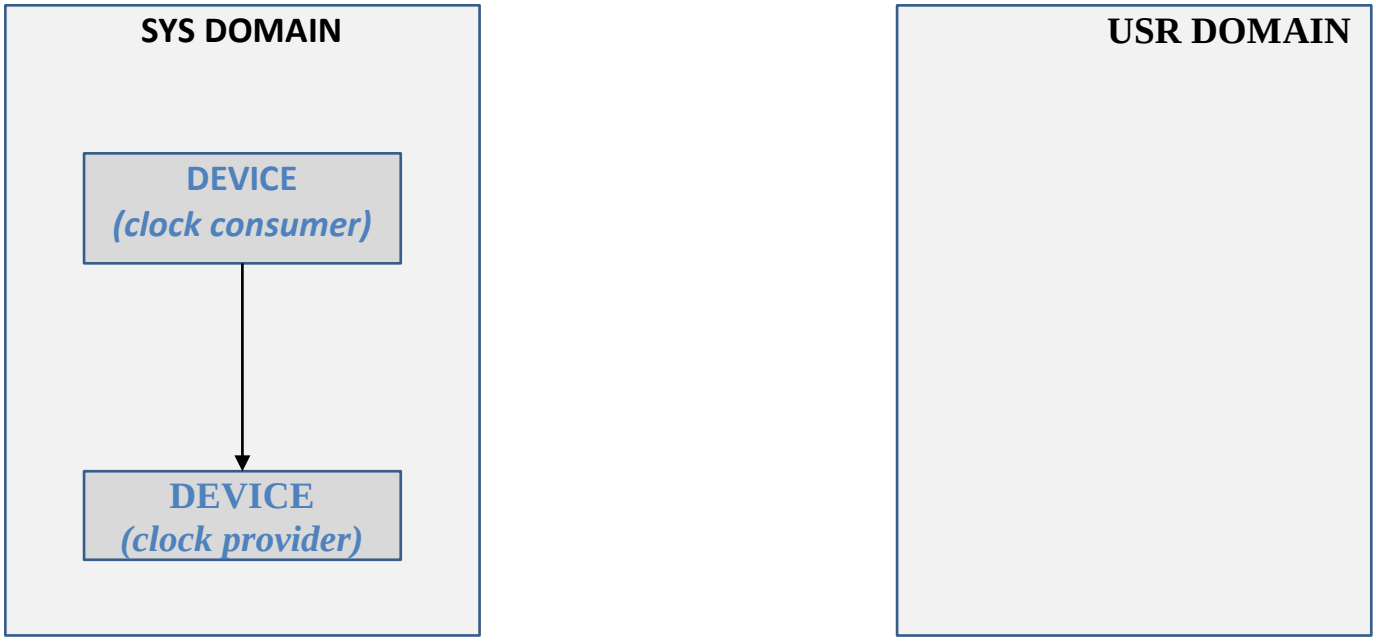


Figure 7-1 - Native Clock Architecture

© 2022 Harman. All Rights Reserved

Confidential

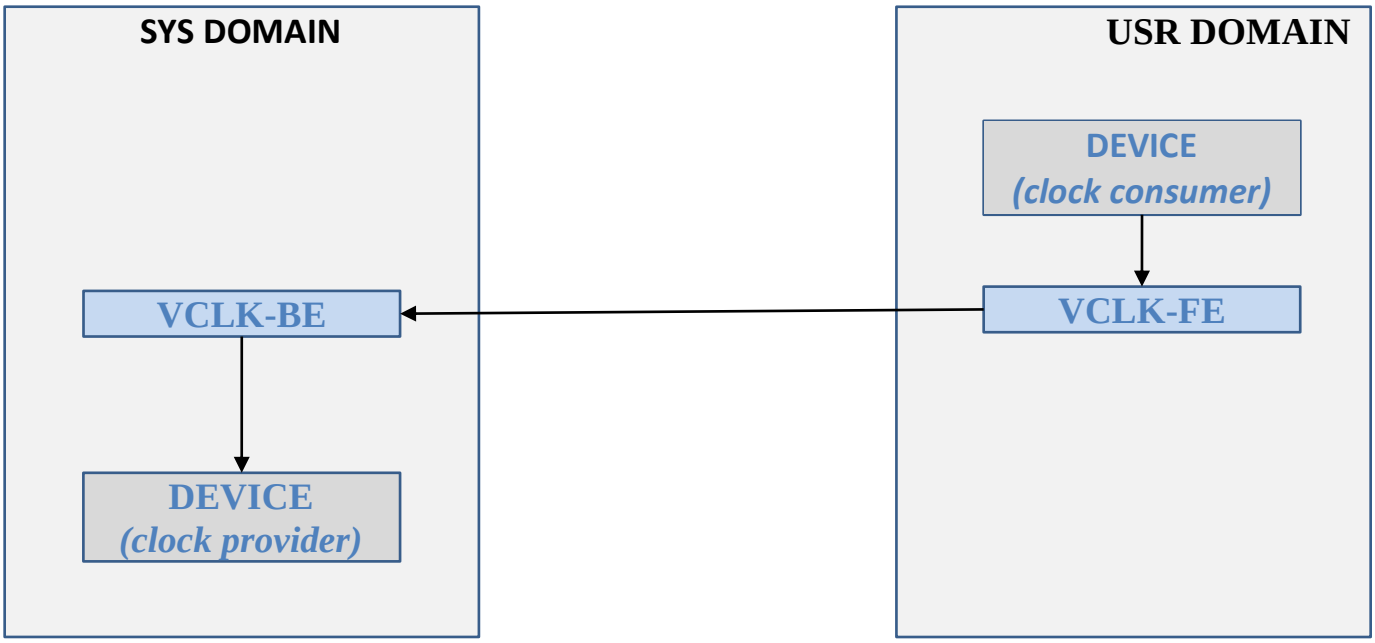


Figure 7-2 - Virtualized Clock Architecture

7.2.1 Enable vclk drivers

Virtual clock back-end driver should be enabled in SYS domain Linux Kernel Configuration:

Device Drivers->VLX virtual device support -> VLX virtual clock control back end driver

Virtual clock front-end driver should be enabled in USR domain Linux Kernel Configuration:

Device Drivers->VLX virtual device support -> VLX virtual clock control front end driver

7.2.2 Setup vRPC

Back-end and front-end drivers communicate using vRPC. Therefore, those nodes should be added to hypervisor's configuration:

```
&vm2_vdevs {
   vclk_be: vclk@be {
      compatible = "vrpc";
      server;
      info = "vclk_ctrl";
   };
};

&vm3_vdevs {
   vclk@fe {
      peer-phandle = <&vclk_be>;
      client;
      info = "vclk_ctrl";
   };
```

© 2022 Harman. All Rights Reserved

Confidential

```
};
```

7.2.3 Back-end Configuration

In SYS domain, Linux DTS has to be modified to configure the vclock back-end driver. This configuration consists in two nodes:

- the original clock consumer node (Exynode ADC): we keep its definition in SYS domain DTS, but we disable it. It is only kept to be referenced by the next new node.
- a new node compatible with “vl,vclk-be”: this node defines all the clock sources exported to a given VM.

Below is an example of such configuration:

```
exynos_adc: adc@13620000 {
   compatible = "disabled";  // Disabled, but kept for parsing.
   reg = <0x0 0x13620000 0x100>;
   interrupts = <0 423 0>;
   #io-channel-cells = <1>;
   io-channel-ranges;
   clocks = <&clock 209>;
   clock-names = "gate_adcif";
};

clk-be@3 {
   compatible = "vl,vclk-be”;

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

   // Exported clock provider.
   vl,clock-providers = <&clock>;

   // Provider ID, as used in front-end DTS.
   vl,clock-provider-names = “clock-be”;

   clk@0 {
      // Limit the export to signals used by this consumer.
      vl,clock-ref = <&exynos_adc>;
   };

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

7.2.4 Front-end Configuration

In USR domain, Linux DTS has to be modified to configure a virtual clock front-end:

```
// Front-end clock provider: compatible with “vl,vclk-fe”, it has the
// same #clock-cell value as the original clock provider and it will be
```

```
// mapped to the back-end based on the value of "vl,clock-provider-name".
clock: clock-controller@0x10570000 {
   compatible = "vl,vclk-fe";
   vl,clock-provider-name = "clock-be";
   #clock-cells = <1>;
};

// Copy as it is of the original clock consumer.
exynos_adc: adc@13620000 {
   compatible = "samsung,exynos-adc-v2";
   reg = <0x0 0x13620000 0x100>;
   interrupts = <0 423 0>;
   #io-channel-cells = <1>;
   io-channel-ranges;
   clocks = <&clock 209>;
   clock-names = "gate_adcif";
};
```

7.2.5 Other features

- This is sometimes useful to always prepare and enable a given clock at back-end startup. This can be done adding an empty `vl,force-enable` property right next to the property `vl,clock-ref` referencing the targeted clock.
- To always prepare and enable all the clocks referenced by the virtual clock back-end configuration node, you can also enable the following Linux configuration:

  Device Drivers -> VLX virtual device support -> Enable all exported clocks at the boot time
- By default, the virtual clock driver is using the RPC channel `vclk_ctrl` to remotely execute all clock operations in a kernel thread of the back-end. There is an exception to this rule: `enable` and `disable` operations are not handled like that because they are supposed to be atomic. In its default implementation, virtual clock driver implements `enable` and `disable` operations as empty callbacks and performs the real `enable/disable` work at the end of the `prepare` and at the beginning of the `unprepare` operations. There might be corner cases where this default behavior causes issues[5], so the user can change it on a per-clock basis switching to a mode that uses a second RPC channel to really execute the `enable/disable` operations on the back-end in interrupt context. To select this mode, the user must define an extra VRPC channel for the targeted (back-end, front-end) pair named `vclk_fast_ctrl`, and add a property named `vl,use-fast-rpc` right next to the `vl,clock-ref` property referencing the targeted clocks.

7.2.6 Limitations

- Virtual clock front-ends are always seen as orphan, so they don't support the operations *`clk_set_parent()`* or *`clk_get_parent()`*.
- A locally cached rate value is checked during *`clk_set_rate()`*. This cached value is not invalidated in the front-end when a consumer of the back-end VM changes the rate. If

[5] When prepare/enable are not done in sequence. Typically, this starts the device earlier and may cause an unexpected behavior

© 2022 Harman. All Rights Reserved

Confidential

multiple VMs compete to set the rate of the same clock, this can lead to one of the `clk_set_rate()` to **not** change the rate as expected.

For more information about virtual clock, please refer to the *VCLK* manual page in the *Virtual Device Driver Reference Manual*.

8 Power Domain Configuration

8.1 Power Domains

Most of the devices of modern hardware platforms, can be powered-on and powered-off. A power controller allows to switch the device on or off. Generally a single switch may power multiple devices, in that case we talk about power-domain. A power domain manager is a device controlling the power-on and power-off states of a device.

Dependency issues can also come from power domain nodes in the device tree. Below is a typical example of a power domain provider and a power domain consumer:

```
// Power domain provider
pd_aud: pd-aud@14064000 {
        compatible = "samsung,exynos-pd";
        reg = <0x0 0x14064000 0x20>;
        #power-domain-cells = <0>;
        …
};

// Power domain consumer
dbgdev-pd-aud: dbgdev-pd-aud@14064000 {
         compatible = "samsung,exynos-pd-dbg";
         power-domains = <&pd_aud>;
};
```

If you move the power domain consumer (device `dbgdev-pd-aud`) as it is from SYS domain to USR domain, you will break the DTS build because of the property `power-domains` that references the provider `pd_aud`. To solve this problem, you will need to virtualize the power domain provider.

8.2 Virtualizing Power Domains

Power domain virtualization allows the `dbgdev-pd-aud` device in USR domain to rely on its original power domain defined in SYS domain device tree, thus solving the dependency issue previously described.

The next two figures illustrate how power domain virtualization works: it is based on a front-end driver and a back-end driver that allow a device owned by the USR domain to still rely on a power domain owned by the SYS domain.

© 2022 Harman. All Rights Reserved

Confidential

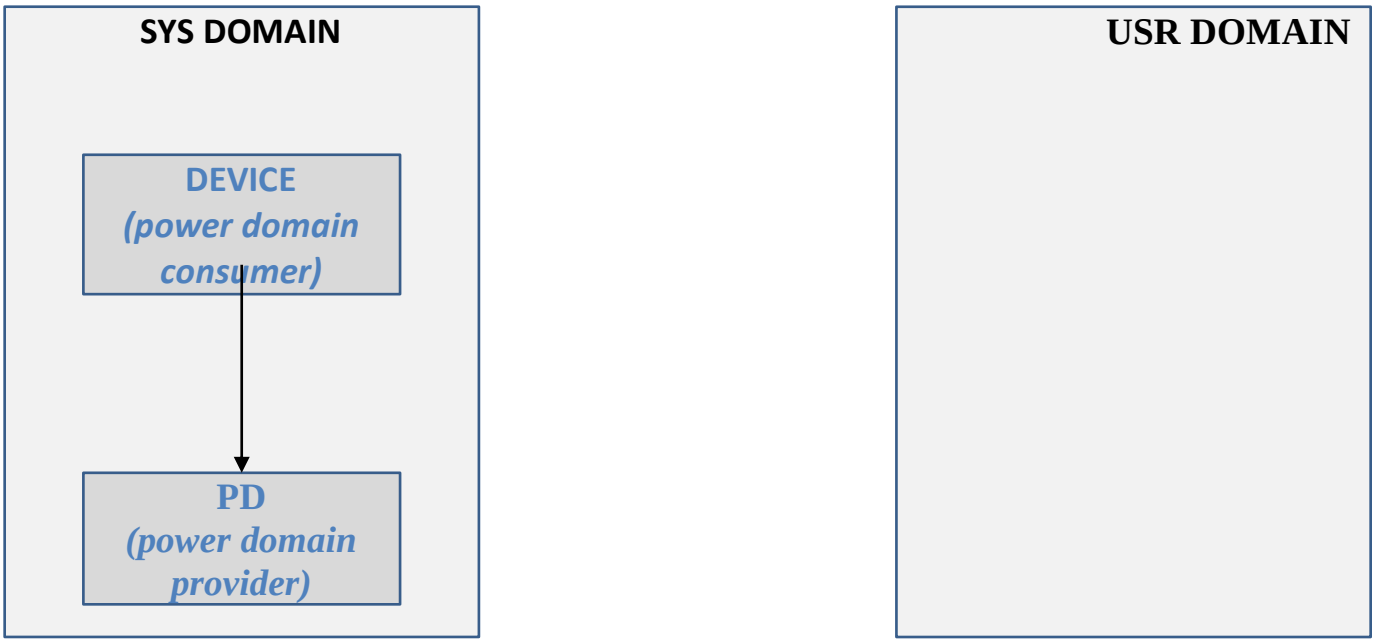


Figure 8-1 - Native Power Domain Architecture

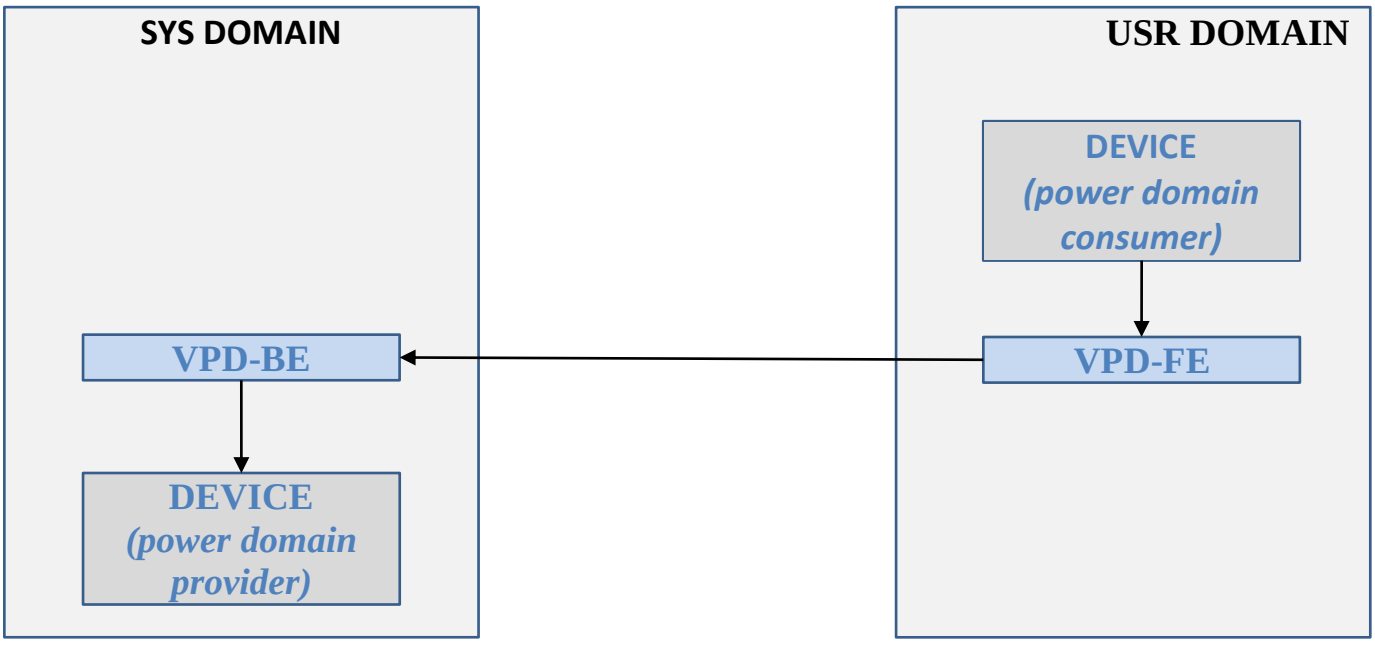


Figure 8-2 – Virtualized Power Domain Architecture

8.2.1 Enable vpd drivers

Virtual power domain back-end driver should be enabled in SYS domain Linux Kernel Configuration:

Device Drivers->VLX virtual device support -> VLX power domain back end driver

Virtual power domain front-end driver should be enabled in USR domain Linux Kernel Configuration:

Device Drivers->VLX virtual device support -> VLX power domain front end driver

8.2.2 Setup vRPC

Back-end and front-end drivers communicate using vRPC. Therefore, those nodes should be added to hypervisor's configuration:

```
&vm2_vdevs {
   vpd_be: vpd@be {
      compatible = "vrpc";
      server;
      info = "vpd_ctrl";
   };
};

&vm3_vdevs {
   vpd@fe {
      peer-phandle = <&vpd_be>;
      client;
      info = "vpd_ctrl";
   };
};
```

8.2.3 Back-end Configuration

In SYS domain, Linux DTS must be modified to configure the virtual power domain back-end driver. This configuration consists in two nodes:

- the original power domain consumer node (`dbgdev-pd-aud`): we keep its definition in SYS domain DTS, but we disable it. It is only kept to be referenced by the next new node.
- a new node compatible with "vl,vpower-domain-be": this node describes all the power domains exported to a given VM.

Below is an example of such configuration:

```
// Power domain consumer, disabled but kept for parsing.
dbgdev-pd-aud: dbgdev-pd-aud@14064000 {
         //compatible = "samsung,exynos-pd-dbg";
         compatible = "disabled";
         power-domains = <&pd_aud>;
};

vpd-be@3 {
   compatible = "vl,vpower-domain-be";

   // This node exports power domains to VM3.
   vl,vm-id = <3>;

   // Exported power domain provider.
```

```
   vl,power-domain-providers = <&pd_aud>;

   // Provider ID, as used in front-end DTS.
   vl,power-domain-provider-names = "pd_aud-be";

   pd@0 {
      // Limit the export to power domains used by this consumer.
      vl,power-domain-ref = <&dbgdev-pd-aud>;
   };

   // You could add other consumers here...
   // pd@1 { vl,power-domain-ref = <&other_pd_consumer>};
}
```

8.2.4 Front-end Configuration

In USR domain, Linux DTS must be modified to configure a virtual power domain front-end:

```
// Front-end power domain provider:
//     compatible with "vl,vpower-domain-fe", it has
//     the same #power-domain-cell value as the original provider and it
//     will be mapped to the back-end based on the value of
//     "vl,power-domain-provider-name".
pd_aud: pd-aud@14064000 {
        compatible = "vl,vpower-domain-fe";
        #power-domain-cells = <0>;
};

// Copy as it is of the original clock consumer.
dbgdev-pd-aud: dbgdev-pd-aud@14064000 {
         compatible = "samsung,exynos-pd-dbg";
         power-domains = <&pd_aud>;
};
```

9 Pin Configuration

9.1 Pin Controller

Due to the limited amount of pins, they might be shared by multiple peripherals. In such a case, the device driver of the migrated device shall configure the related pin. This chapter describes this configuration operation.

Under Linux, hardware modules that control pin multiplexing or parameters such as pull-up / pull-down, tri-state, or drive strength are known as **pin controllers**. A pin controller is typically described in the Linux device tree by a node that includes the usual `reg` property describing the range of registers that control pin configuration and a list of sub-nodes, each sub-node describing a possible configuration for a set of pins. Below is an example of such a description:

```
pinctrl_5: pinctrl@10230000 {
   compatible = "samsung,exynosautov9-pinctrl";
```

© 2022 Harman. All Rights Reserved

Confidential

```
   reg = <0x0 0x10230000 0x1000>;

   spi1_bus: spi1-bus {
      /* state description */
      samsung,pins = "gpp0-6", "gpp0-5", "gpp0-4";
      samsung,pin-function = <2>;
      samsung,pin-pud = <0>;
      samsung,pin-drv = <0>;
   };
};
```

The state `spi1_bus` is defined by a few Samsung specific properties, but we can easily guess that this state configures several pins: GPP0-4, GPP0-5, and GPP0-6.

To work properly, a device can require some pin configuration. This constraint is described in the device tree using properties `pinctrl-<index>` and `pinctrl-names` as seen below:

```
spi_1: spi@10320000 {
   compatible = "samsung,exynos-spi";
   reg = <0x0 0x10320000 0x100>;
   /* .. */
   pinctrl-names = "default";
   pinctrl-0 = <&spi1_bus>;
};
```

Above node describes an SPI bus controller device that depends on the pin configuration known as `default` that is defined by the `spi1_bus` node. The `default` pin configuration is a special configuration in that it is automatically performed by the Linux framework just before the kernel calls the `probe()` function of the driver. In other words, Linux will configure pins according to `spi1_bus` state in the early steps of SPI controller driver initialization.

The previous example shows that a device can depend on a pin controller to setup pin configuration. This can lead to a new kind of dependency issue when you split your devices among multiple domains: if two different devices depend on the same pin controller and you want to only move one of the devices to another domain, you will need a new virtualization mechanism. This new mechanism is described in the next section.

9.2 Splitting pin configuration between VMs

Pin control can lead to a dependency issue when you try to move one device from SYS domain to USR domain. To solve this problem, below are the steps you should follow:

- identify the responsibility of each VM:
  - identify which VM should perform each pin controller state configuration
  - in SoC datasheet (or Linux kernel driver), identify pin controller registers, and what VM should own each register
- **move** your device from SYS domain to USR domain
- **copy** the pin controller that your device depends on to the USR domain.
- install and configure **virtual GPIOs** on USR domain. Those virtual GPIOs will define the exact registers of pin controller that USR domain can access.

© 2022 Harman. All Rights Reserved

Confidential

- install and configure **virtual GPIOs** on SYS domain. Those virtual GPIOs will define the exact registers of pin controller that SYS domain can access.

Let's suppose that we want to move the SPI controller device `spi_1` seen in the example of section 9.1 from SYS domain to USR domain. USR domain will be responsible for configuring pin controller state `spi1_bus`, and SYS domain will keep the responsibility of controlling all the other pin controller states. Registers controlling GPP0-4, GPP0-5, GPP0-6 are the ones involved in `spi1_bus` state, so they should be owned by USR domain.

Virtual GPIO configuration is located in virtual platform (vplatform) configuration of a given VM. It is composed of three kind of device tree nodes. Below is a USR domain virtual GPIO configuration that allows USR domain to control GPP0-4, GPP0-5, and GPP0-6 pins:

```
// vGPIO overall config node
vl,vgpio@10230000 {
   compatible = "vl,vgpio";
   reg = <0x0 0x10230000 0x1000>;
   write-enable-range = <&gpp0 4 6>;
};

// vGPIO bank node
gpp0: gpp0 {
   compatible = "vl,vgpio-bank";
   #count = <8>;
   offset = <0x0>;
   layout = <&gpio_off>;
};

// vGPIO layout node
gpio_off: gpio-bank-off {
   compatible = "vl,vgpio-layout";
   reg-offset = <0x00 0x04 0x08 0x0c 0x10 0x14>;
   field-width = <4 1 4 4 2 4>;
};
```

In the above example, the first configuration node, compatible with `vl,vgpio`, specifies a range of registers using the `reg` property. If no other property is specified, this node will instruct the hypervisor to only provide read-access to USR domain on this given range of addresses. It means any write access will be ignored. This default behavior can be modified adding the property `write-enable-range` to provide write permissions to a set of pins. Those are specified using a reference to a GPIO bank and a range of indexes. The above example gives write access to pins 4, 5, 6 in the bank `gpp0`.

A GPIO bank is defined by a node compatible with `vl,vgpio-bank`. In this node, the property `#count` specifies the number of pins in the bank and the `offset` property specifies the start offset of bank's registers from the base address specified in the `reg` property of the previous node. The `layout` property references a GPIO layout node, which is described in the next paragraph.

© 2022 Harman. All Rights Reserved

Confidential

The last node, compatible with `vl,vgpio-layout`, describes the layout of 32-bit registers controlling pins. The property `reg-offset` specifies the register offsets from the bank base address. The `field-width` property specifies, for each register, the number of bits that control a single pin. By default, all those bits are given write access, except if an optional property `field-mask` is used to specify the exact bits that the GuestOS will be allowed to write.

Below is a table that lists, for each gpp0 bank register, the bits that write permission is given:

	0x10230000	**0x10230004**	**0x10230008**	**0x1023000c**	**0x10230010**	**0x10230014**
GPIO 4	0x000F0000	0x00000010	0x000F0000	0x000F0000	0x00000300	0x000F0000
GPIO 5	0x00F00000	0x00000020	0x00F00000	0x00F00000	0x00000c00	0x00F00000
GPIO 6	0x0F000000	0x00000040	0x0F000000	0x0F000000	0x00003000	0x0F000000
Overall write permission mask for USR domain	0x0FFF0000	0x00000070	0x0FFF0000	0x0FFF0000	0x00003F00	0x0FFF0000

Now that USR domain has been configured, you still need to configure SYS domain. Let's suppose that you want to let SYS domain access all the registers of bank `gpp0`, but the ones given to USR domain. The following nodes will configure those restrictions:

```
// vGPIO overall config node
vl,vgpio@10230000 {
   compatible = "vl,vgpio";
   reg = <0x0 0x10230000 0x1000>;
   write-enable-range = <&gpp0 0 7>;
   write-disable-range = <&gpp0 4 6>;
};

// vGPIO bank node
gpp0: gpp0 {
   compatible = "vl,vgpio-bank";
   #count = <8>;
   offset = <0x0>;
   layout = <&gpio_off>;
};

// vGPIO layout node
gpio_off: gpio-bank-off {
   compatible = "vl,vgpio-layout";
   reg-offset = <0x00 0x04 0x08 0x0c 0x10 0x14>;
   field-width = <4 1 4 4 2 4>;
};
```

© 2022 Harman. All Rights Reserved

Confidential

This configuration is very close to the one specified for USR domain, except for its first node, that enables writes to all GPIOs in the bank `gpp0`, but GPIOs 4, 5, and 6.

For more information about virtual GPIO configuration, please refer to the section *Configuration > VM Virtual Platform bindings > Virtual GPIO Node* in the *Hypervisor Reference Manual*.

10 Other Potential Issues

When you move a device from SYS Domain to USR Domain, you might also encounter other issues related:

- GPIO interrupt handling
- to DMA configuration

that are not covered by this version of the document.

© 2022 Harman. All Rights Reserved

Confidential