VFIO Mediated devices

Author: Neo Jia <cjia@nvidia.com>
Author: Kirti Wankhede <kwankhede@nvidia.com>

Virtual Function I/O (VFIO) Mediated devices[1]

The number of use cases for virtualizing DMA devices that do not have built-in SR_IOV capability is increasing. Previously, to virtualize such devices, developers had to create their own management interfaces and APIs, and then integrate them with user space software. To simplify integration with user space software, we have identified common requirements and a unified management interface for such devices.

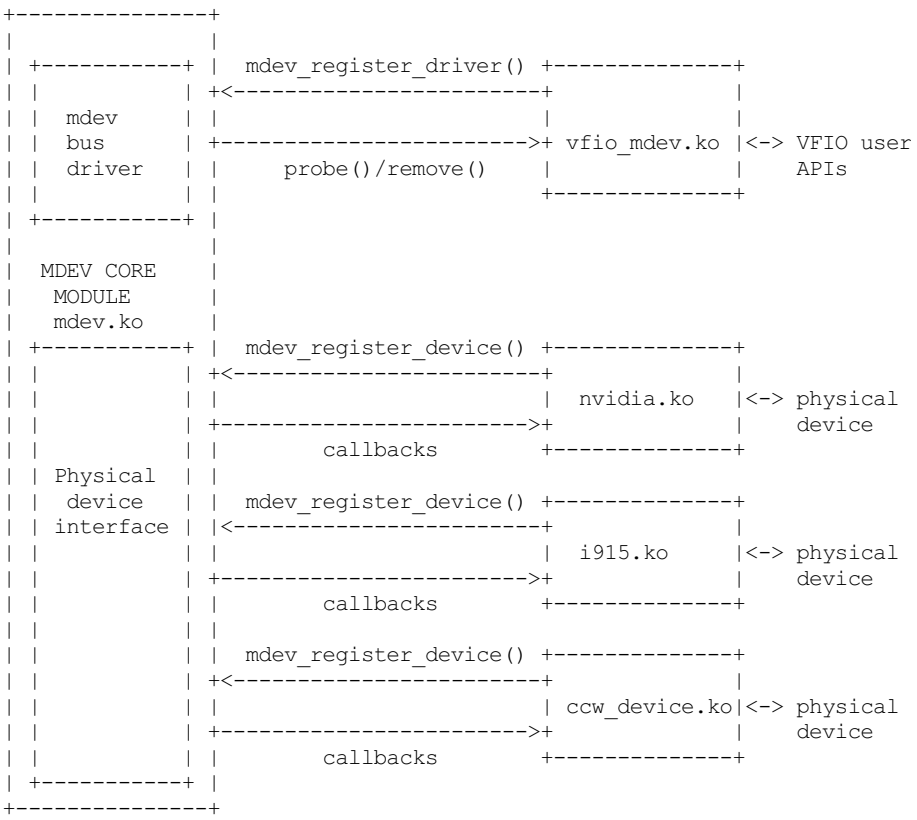
The VFIO driver framework provides unified APIs for direct device access. It is an IOMMU/device-agnostic framework for exposing direct device access to user space in a secure, IOMMU-protected environment. This framework is used for multiple devices, such as GPUs, network adapters, and compute accelerators. With direct device access, virtual machines or user space applications have direct access to the physical device. This framework is reused for mediated devices.

The mediated core driver provides a common interface for mediated device management that can be used by drivers of different devices. This module provides a generic interface to perform these operations:

- Create and destroy a mediated device
- Add a mediated device to and remove it from a mediated bus driver
- Add a mediated device to and remove it from an IOMMU group

The mediated core driver also provides an interface to register a bus driver. For example, the mediated VFIO mdev driver is designed for mediated devices and supports VFIO APIs. The mediated bus driver adds a mediated device to and removes it from a VFIO group.

The following high-level block diagram shows the main components and interfaces in the VFIO mediated driver framework. The diagram shows NVIDIA, Intel, and IBM devices as examples, as these devices are the first devices to use this module:



Registration Interfaces

The mediated core driver provides the following types of registration interfaces:

- Registration interface for a mediated bus driver
- Physical device driver interface

Registration Interface for a Mediated Bus Driver

The registration interface for a mediated device driver provides the following structure to represent a mediated device's driver:

```
/*
 * struct mdev_driver [2] - Mediated device's driver
 * @probe: called when new device created
 * @remove: called when device removed
 * @driver: device driver structure
 */
struct mdev_driver {
        int  (*probe)  (struct mdev_device *dev);
        void (*remove) (struct mdev_device *dev);
        struct device_driver    driver;
};
```

A mediated bus driver for mdev should use this structure in the function calls to register and unregister itself with the core driver:

- Register:

```
extern int  mdev_register_driver(struct mdev_driver *drv);
```

- Unregister:

```
extern void mdev_unregister_driver(struct mdev_driver *drv);
```

The mediated bus driver is responsible for adding mediated devices to the VFIO group when devices are bound to the driver and removing mediated devices from the VFIO when devices are unbound from the driver.

Physical Device Driver Interface

The physical device driver interface provides the mdev_parent_ops[3] structure to define the APIs to manage work in the mediated core driver that is related to the physical device.

The structures in the mdev_parent_ops structure are as follows:

- dev_attr_groups: attributes of the parent device
- mdev_attr_groups: attributes of the mediated device
- supported_config: attributes to define supported configurations
- device_driver: device driver to bind for mediated device instances

The mdev_parent_ops also still has various functions pointers. Theses exist for historical reasons only and shall not be used for new drivers.

When a driver wants to add the GUID creation sysfs to an existing device it has probe'd to then it should call:

```
extern int  mdev_register_device(struct device *dev,
                                 const struct mdev_parent_ops *ops);
```

This will provide the 'mdev_supported_types/XX/create' files which can then be used to trigger the creation of a mdev_device. The created mdev_device will be attached to the specified driver.

When the driver needs to remove itself it calls:

```
extern void mdev_unregister_device(struct device *dev);
```

Which will unbind and destroy all the created mdevs and remove the sysfs files.

Mediated Device Management Interface Through sysfs

The management interface through sysfs enables user space software, such as libvirt, to query and configure mediated devices in a hardware-agnostic fashion. This management interface provides flexibility to the underlying physical device's driver to support features such as:

- Mediated device hot plug
- Multiple mediated devices in a single virtual machine
- Multiple mediated devices from different physical devices

Links in the mdev_bus Class Directory

The /sys/class/mdev_bus/ directory contains links to devices that are registered with the mdev core driver.

Directories and files under the sysfs for Each Physical Device

```
|- [parent physical device]
|--- Vendor-specific-attributes [optional]
|--- [mdev_supported_types]
|     |--- [<type-id>]
```

```
|      |   |--- create
|      |   |--- name
|      |   |--- available_instances
|      |   |--- device_api
|      |   |--- description
|      |   |--- [devices]
|      |--- [<type-id>]
|      |   |--- create
|      |   |--- name
|      |   |--- available_instances
|      |   |--- device_api
|      |   |--- description
|      |   |--- [devices]
|      |--- [<type-id>]
|          |--- create
|          |--- name
|          |--- available_instances
|          |--- device_api
|          |--- description
|          |--- [devices]
```

- [mdev_supported_types]

  The list of currently supported mediated device types and their details.

  [<type-id>], device_api, and available_instances are mandatory attributes that should be provided by vendor driver.

- [<type-id>]

  The [<type-id>] name is created by adding the device driver string as a prefix to the string provided by the vendor driver. This format of this name is as follows:

  ```
  sprintf(buf, "%s-%s", dev_driver_string(parent->dev), group->name);
  ```

  (or using mdev_parent_dev(mdev) to arrive at the parent device outside of the core mdev code)

- device_api

  This attribute should show which device API is being created, for example, "vfio-pci" for a PCI device.

- available_instances

  This attribute should show the number of devices of type <type-id> that can be created.

- [device]

  This directory contains links to the devices of type <type-id> that have been created.

- name

  This attribute should show human readable name. This is optional attribute.

- description

  This attribute should show brief features/description of the type. This is optional attribute.

Directories and Files Under the sysfs for Each mdev Device

```
|- [parent phy device]
|--- [$MDEV_UUID]
       |--- remove
       |--- mdev_type {link to its type}
       |--- vendor-specific-attributes [optional]
```

- remove (write only)

Writing '1' to the 'remove' file destroys the mdev device. The vendor driver can fail the remove() callback if that device is active and the vendor driver doesn't support hot unplug.

Example:

```
# echo 1 > /sys/bus/mdev/devices/$mdev_UUID/remove
```

Mediated device Hot plug

Mediated devices can be created and assigned at runtime. The procedure to hot plug a mediated device is the same as the procedure to hot plug a PCI device.

Translation APIs for Mediated Devices

The following APIs are provided for translating user pfn to host pfn in a VFIO driver:

```
extern int vfio_pin_pages(struct device *dev, unsigned long *user_pfn,
```

```
                            int npage, int prot, unsigned long *phys_pfn);

extern int vfio_unpin_pages(struct device *dev, unsigned long *user_pfn,
                            int npage);
```

These functions call back into the back-end IOMMU module by using the pin_pages and unpin_pages callbacks of the struct vfio_iommu_driver_ops[4]. Currently these callbacks are supported in the TYPE1 IOMMU module. To enable them for other IOMMU backend modules, such as PPC64 sPAPR module, they need to provide these two callback functions.

Using the Sample Code

mtty.c in samples/vfio-mdev/ directory is a sample driver program to demonstrate how to use the mediated device framework.

The sample driver creates an mdev device that simulates a serial port over a PCI card.

1. Build and load the mtty.ko module.

   This step creates a dummy device, /sys/devices/virtual/mtty/mtty/

   Files in this device directory in sysfs are similar to the following:

   ```
   # tree /sys/devices/virtual/mtty/mtty/
      /sys/devices/virtual/mtty/mtty/
      |-- mdev_supported_types
      |   |-- mtty-1
      |   |   |-- available_instances
      |   |   |-- create
      |   |   |-- device_api
      |   |   |-- devices
      |   |   `-- name
      |   `-- mtty-2
      |       |-- available_instances
      |       |-- create
      |       |-- device_api
      |       |-- devices
      |       `-- name
      |-- mtty_dev
      |   `-- sample_mtty_dev
      |-- power
      |   |-- autosuspend_delay_ms
      |   |-- control
      |   |-- runtime_active_time
      |   |-- runtime_status
      |   `-- runtime_suspended_time
      |-- subsystem -> ../../../../class/mtty
      `-- uevent
   ```

2. Create a mediated device by using the dummy device that you created in the previous step:

   ```
   # echo "83b8f4f2-509f-382f-3c1e-e6bfe0fa1001" >    \
           /sys/devices/virtual/mtty/mtty/mdev_supported_types/mtty-2/create
   ```

3. Add parameters to qemu-kvm:

   ```
   -device vfio-pci,\
    sysfsdev=/sys/bus/mdev/devices/83b8f4f2-509f-382f-3c1e-e6bfe0fa1001
   ```

4. Boot the VM.

   In the Linux guest VM, with no hardware on the host, the device appears as follows:

   ```
   # lspci -s 00:05.0 -xxvv
   00:05.0 Serial controller: Device 4348:3253 (rev 10) (prog-if 02 [16550])
           Subsystem: Device 4348:3253
           Physical Slot: 5
           Control: I/O+ Mem- BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr-
   Stepping- SERR- FastB2B- DisINTx-
           Status: Cap- 66MHz- UDF- FastB2B- ParErr- DEVSEL=medium >TAbort-
   <TAbort- <MAbort- >SERR- <PERR- INTx-
           Interrupt: pin A routed to IRQ 10
           Region 0: I/O ports at c150 [size=8]
           Region 1: I/O ports at c158 [size=8]
           Kernel driver in use: serial
   00: 48 43 53 32 01 00 00 02 10 02 00 07 00 00 00 00
   10: 51 c1 00 00 59 c1 00 00 00 00 00 00 00 00 00 00
   20: 00 00 00 00 00 00 00 00 00 00 00 00 48 43 53 32
   30: 00 00 00 00 00 00 00 00 00 00 00 00 0a 01 00 00

   In the Linux guest VM, dmesg output for the device is as follows:

   serial 0000:00:05.0: PCI INT A -> Link[LNKA] -> GSI 10 (level, high) -> IRQ 10
   ```

```
0000:00:05.0: ttyS1 at I/O 0xc150 (irq = 10) is a 16550A
0000:00:05.0: ttyS2 at I/O 0xc158 (irq = 10) is a 16550A
```

5. In the Linux guest VM, check the serial ports:

```
# setserial -g /dev/ttyS*
/dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4
/dev/ttyS1, UART: 16550A, Port: 0xc150, IRQ: 10
/dev/ttyS2, UART: 16550A, Port: 0xc158, IRQ: 10
```

6. Using minicom or any terminal emulation program, open port /dev/ttyS1 or /dev/ttyS2 with hardware flow control disabled.
7. Type data on the minicom terminal or send data to the terminal emulation program and read the data.

   Data is loop backed from hosts mtty driver.
8. Destroy the mediated device that you created:

```
# echo 1 > /sys/bus/mdev/devices/83b8f4f2-509f-382f-3c1e-e6bfe0fa1001/remove
```

References

1. See Documentation/driver-api/vfio.rst for more information on VFIO.
2. struct mdev_driver in include/linux/mdev.h
3. struct mdev_parent_ops in include/linux/mdev.h
4. struct vfio_iommu_driver_ops in include/linux/vfio.h