Accessing PCI device resources through sysfs

sysfs, usually mounted at /sys, provides access to PCI resources on platforms that support it. For example, a given bus might look like this:

```
/sys/devices/pci0000:17
|-- 0000:17:00.0
|   |-- class
|   |-- config
|   |-- device
|   |-- enable
|   |-- irq
|   |-- local_cpus
|   |-- remove
|   |-- resource
|   |-- resource0
|   |-- resource1
|   |-- resource2
|   |-- revision
|   |-- rom
|   |-- subsystem_device
|   |-- subsystem_vendor
|   `-- vendor
`-- ...
```

The topmost element describes the PCI domain and bus number. In this case, the domain number is 0000 and the bus number is 17 (both values are in hex). This bus contains a single function device in slot 0. The domain and bus numbers are reproduced for convenience. Under the device directory are several files, each with their own function.

file	function
class	PCI class (ascii, ro)
config	PCI config space (binary, rw)
device	PCI device (ascii, ro)
enable	Whether the device is enabled (ascii, rw)
irq	IRQ number (ascii, ro)
local_cpus	nearby CPU mask (cpumask, ro)
remove	remove device from kernel's list (ascii, wo)
resource	PCI resource host addresses (ascii, ro)
resource0..N	PCI resource N, if present (binary, mmap, rw[1])
resource0_wc..N_wc	PCI WC map resource N, if prefetchable (binary, mmap)
revision	PCI revision (ascii, ro)
rom	PCI ROM resource, if present (binary, ro)
subsystem_device	PCI subsystem device (ascii, ro)
subsystem_vendor	PCI subsystem vendor (ascii, ro)
vendor	PCI vendor (ascii, ro)

```
ro - read only file
rw - file is readable and writable
wo - write only file
mmap - file is mmapable
ascii - file contains ascii text
binary - file contains binary data
cpumask - file contains a cpumask type
```

[1] rw for IORESOURCE_IO (I/O port) regions only

The read only files are informational, writes to them will be ignored, with the exception of the 'rom' file. Writable files can be used to perform actions on the device (e.g. changing config space, detaching a device). mmapable files are available via an mmap of the file at offset 0 and can be used to do actual device programming from userspace. Note that some platforms don't support mmapping of certain resources, so be sure to check the return value from any attempted mmap. The most notable of these are I/O port resources, which also provide read/write access.

The 'enable' file provides a counter that indicates how many times the device has been enabled. If the 'enable' file currently returns '4', and a '1' is echoed into it, it will then return '5'. Echoing a '0' into it will decrease the count. Even when it returns to 0, though, some of the initialisation may not be reversed.

The 'rom' file is special in that it provides read-only access to the device's ROM file, if available. It's disabled by default, however, so applications should write the string "1" to the file to enable it before attempting a read call, and disable it following the access by writing "0" to the file. Note that the device must be enabled for a rom read to return data successfully. In the event a driver is not bound to the device, it can be enabled using the 'enable' file, documented above.

The 'remove' file is used to remove the PCI device, by writing a non-zero integer to the file. This does not involve any kind of hot-plug functionality, e.g. powering off the device. The device is removed from the kernel's list of PCI devices, the sysfs directory for it is removed, and the device will be removed from any drivers attached to it. Removal of PCI root buses is disallowed.

Accessing legacy resources through sysfs

Legacy I/O port and ISA memory resources are also provided in sysfs if the underlying platform supports them. They're located in the PCI class hierarchy, e.g.:

```
/sys/class/pci_bus/0000:17/
|-- bridge -> ../../../devices/pci0000:17
|-- cpuaffinity
|-- legacy_io
`-- legacy_mem
```

The legacy_io file is a read/write file that can be used by applications to do legacy port I/O. The application should open the file, seek to the desired port (e.g. 0x3e8) and do a read or a write of 1, 2 or 4 bytes. The legacy_mem file should be mmapped with an offset corresponding to the memory offset desired, e.g. 0xa0000 for the VGA frame buffer. The application can then simply dereference the returned pointer (after checking for errors of course) to access legacy memory space.

Supporting PCI access on new platforms

In order to support PCI resource mapping as described above, Linux platform code should ideally define ARCH_GENERIC_PCI_MMAP_RESOURCE and use the generic implementation of that functionality. To support the historical interface of mmap() through files in /proc/bus/pci, platforms may also set HAVE_PCI_MMAP.

Alternatively, platforms which set HAVE_PCI_MMAP may provide their own implementation of pci_mmap_page_range() instead of defining ARCH_GENERIC_PCI_MMAP_RESOURCE.

Platforms which support write-combining maps of PCI resources must define arch_can_pci_mmap_wc() which shall evaluate to non-zero at runtime when write-combining is permitted. Platforms which support maps of I/O resources define arch_can_pci_mmap_io() similarly.

Legacy resources are protected by the HAVE_PCI_LEGACY define. Platforms wishing to support legacy functionality should define it and provide pci_legacy_read, pci_legacy_write and pci_mmap_legacy_page_range functions.