VGA Arbiter
==========

Graphic devices are accessed through ranges in I/O or memory space. While most modern devices allow relocation of such ranges, some "Legacy" VGA devices implemented on PCI will typically have the same "hard-decoded" addresses as they did on ISA. For more details see "PCI Bus Binding to IEEE Std 1275-1994 Standard for Boot (Initialization Configuration) Firmware Revision 2.1" Section 7, Legacy Devices.

The Resource Access Control (RAC) module inside the X server [0] existed for the legacy VGA arbitration task (besides other bus management tasks) when more than one legacy device co-exists on the same machine. But the problem happens when these devices are trying to be accessed by different userspace clients (e.g. two server in parallel). Their address assignments conflict. Moreover, ideally, being an userspace application, it is not the role of the the X server to control bus resources. Therefore an arbitration scheme outside of the X server is needed to control the sharing of these resources. This document introduces the operation of the VGA arbiter implemented for Linux kernel.

I. Details and Theory of Operation
==========

I.1 vgaarb

The vgaarb is a module of the Linux Kernel. When it is initially loaded, it scans all PCI devices and adds the VGA ones inside the arbitration. The arbiter then enables/disables the decoding on different devices of the VGA legacy instructions. Device which do not want/need to use the arbiter may explicitly tell it by calling vga_set_legacy_decoding().

The kernel exports a char device interface (/dev/vga_arbiter) to the clients, which has the following semantics:

 open   : open user instance of the arbiter. By default, it's attached to
              the default VGA device of the system.

 close   : close user instance. Release locks made by the user

 read   : return a string indicating the status of the target like:

              "<card_ID>,decodes=<io_state>,owns=<io_state>,locks=<io_state>
(ic,mc)"

An IO state string is of the form {io,mem,io+mem,none}, mc and ic are respectively mem and io lock counts (for debugging/ diagnostic only). "decodes" indicate what the card currently decodes, "owns" indicates what is currently enabled on it, and "locks" indicates what is locked by this card. If the card is unplugged, we get "invalid" then for card_ID and an -ENODEV error is returned for any command until a new card is targeted.

write : write a command to the arbiter. List of commands:

target <card_ID> : switch target to card <card_ID> (see below)
lock <io_state> : acquires locks on target ("none" is an invalid io_state)
trylock <io_state> : non-blocking acquire locks on target (returns EBUSY if unsuccessful)
unlock <io_state> : release locks on target
unlock all : release all locks on target held by this user (not implemented yet)
decodes <io_state> : set the legacy decoding attributes for the card

poll : event if something changes on any card (not just the target)

card_ID is of the form "PCI:domain:bus:dev.fn". It can be set to "default" to go back to the system default card (TODO: not implemented yet). Currently, only PCI is supported as a prefix, but the userland API may support other bus types in the future, even if the current kernel implementation doesn't.

Note about locks:

The driver keeps track of which user has which locks on which card. It supports stacking, like the kernel one. This complexifies the implementation a bit, but makes the arbiter more tolerant to user space problems and able to properly cleanup in all cases when a process dies.
Currently, a max of 16 cards can have locks simultaneously issued from user space for a given user (file descriptor instance) of the arbiter.

In the case of devices hot-{un,}plugged, there is a hook - pci_notify() - to notify them being added/removed in the system and automatically added/removed in the arbiter.

There's also a in-kernel API of the arbiter in the case of DRM, vgacon and others which may use the arbiter.

I.2 libpciaccess

To use the vga arbiter char device it was implemented an API inside the libpciaccess library. One field was added to struct pci_device (each device on the system):

```
/* the type of resource decoded by the device */
int vgaarb_rsrc;
```

Besides it, in pci_system were added:

```
    int vgaarb_fd;
    int vga_count;
    struct pci_device *vga_target;
    struct pci_device *vga_default_dev;
```

The vga_count is usually need to keep informed how many cards are being arbitrated, so for instance if there's only one then it can totally escape the scheme.

These functions below acquire VGA resources for the given card and mark those resources as locked. If the resources requested are "normal" (and not legacy) resources, the arbiter will first check whether the card is doing legacy decoding for that type of resource. If yes, the lock is "converted" into a legacy resource lock. The arbiter will first look for all VGA cards that might conflict and disable their IOs and/or Memory access, including VGA forwarding on P2P bridges if necessary, so that the requested resources can be used. Then, the card is marked as locking these resources and the IO and/or Memory access is enabled on the card (including VGA forwarding on parent P2P bridges if any). In the case of vga_arb_lock(), the function will block if some conflicting card is already locking one of the required resources (or any resource on a different bus segment, since P2P bridges don't differentiate VGA memory and IO afaik). If the card already owns the resources, the function succeeds. vga_arb_trylock() will return (-EBUSY) instead of blocking. Nested calls are supported (a per-resource counter is maintained).

Set the target device of this client.

```
    int  pci_device_vgaarb_set_target   (struct pci_device *dev);
```

For instance, in x86 if two devices on the same bus want to lock different resources, both will succeed (lock). If devices are in different buses and trying to lock different resources, only the first who tried succeeds.

```
    int  pci_device_vgaarb_lock         (void);
    int  pci_device_vgaarb_trylock      (void);
```

Unlock resources of device.

```
    int  pci_device_vgaarb_unlock       (void);
```

Indicates to the arbiter if the card decodes legacy VGA IOs, legacy VGA Memory, both, or none. All cards default to both, the card driver (fbdev for example) should tell the arbiter if it has disabled legacy decoding, so the card can be left out of the arbitration process (and can be safe to take interrupts at any time.

```
    int  pci_device_vgaarb_decodes      (int new_vgaarb_rsrc);
```

Connects to the arbiter device, allocates the struct

```
    int  pci_device_vgaarb_init         (void);
```

Close the connection

```
    void pci_device_vgaarb_fini         (void);
```

I.3 xf86VGAArbiter (X server implementation)

(TODO)

X server basically wraps all the functions that touch VGA registers somehow.

II. Credits

Benjamin Herrenschmidt (IBM?) started this work when he discussed such design with the Xorg community in 2005 [1, 2]. In the end of 2007, Paulo Zanoni and Tiago Vignatti (both of C3SL/Federal University of Paran? proceeded his work enhancing the kernel code to adapt as a kernel module and also did the implementation of the user space side [3]. Now (2009) Tiago Vignatti and Dave Airlie finally put this work in shape and queued to Jesse Barnes' PCI tree.

III. References

[0] http://cgit.freedesktop.org/xorg/xserver/commit/?id=4b42448a2388d40f257774fbffdccaea87bd0347
[1] http://lists.freedesktop.org/archives/xorg/2005-March/006663.html
[2] http://lists.freedesktop.org/archives/xorg/2005-March/006745.html
[3] http://lists.freedesktop.org/archives/xorg/2007-October/029507.html