Driver Domain Bring-Up Under Xen-powered Environment on ARM Platforms

In the following article I want to describe GlobalLogic's team experience of bring-up a driver domain on ARM platform. The results that we have achieved with a driver domain were introduced at CES 2015 show and GENIVI event.

What is a driver domain and why it is needed

This article follows the topic described in **Device Passthrough to Driver Domain in Xen** - http://wiki.xenproject.org/wiki/File:Device_passthrough_xen.pdf - where basic concepts of driver domains are described together with hardware passthrough to guest domains.

Driver domains on Nautilus platform

In this article I will refer to the Nautilus solution, which we are currently developing. It successfully runs three driver domains under a Xen hypervisor:
- Cluster domain, which runs automotive-related stuff, manages a hardware video camera and CAN. Based on Linux 3.14.
- Linux driver domain, which controls almost all hardware, runs PV drivers and backends drivers. Based on Linux 3.14.
- Android KitKat guest domain, which manages a hardware video decoder (an image processor unit, also known as IPU remote processor). It is used for infotainment purposes, such as Internet browsing, audio playback, etc.

Driver domains on SMMU-less platforms

One of the most important issues, which has to be solved during a driver domain bring-up is DMA access in driver domain. This strictly depends on SMMU / IOMMU availability. If it is installed, DMA operations may be used securely in a driver domain, otherwise the only way is to use 1:1 physical-to-machine memory mapping in a driver domain the same way as in domain 0. Jacinto6, which currently runs Nautilus, doesn't have SMMU, so a driver domain on it is mapped 1:1. Also, as in domain 0, a SWIOTLB feature has to be enabled (https://blog.xenproject.org/2013/08/14/swiotlb-by-morpheus) and xen_dma_ops has to be used for DMA operations.

Steps to bring-up a driver domain: GlobalLogic experience

- Prepare a domain config file for xl create tool. In this file it is necessary to specify all iomem regions and interrupts, which will be mapped to a driver domain for the following usage in platform drivers. Typically, these resources are similar to the ones defined in a device tree blob.

- Adjust xenpolicy. A driver domain is different from a trusted domain 0 and a guest domain U, so it is a good idea to create a standalone security label for it and assign proper security rights. For example, a driver domain must have access to real hardware, so corresponding rules which allows access to iomem and irqs have to be added.

```
+admin_device(domd_t, device_t)
+admin_device(domd_t, irq_t)
+admin_device(domd_t, ioport_t)
+admin_device(domd_t, iomem_t)
```

- Adjust hypervisor. Added changes are necessary to map a driver domain physical memory 1:1 to a machine memory and IRQ passthrough.

- Adjust xentools. Changes are minimal here. The possibility of allocation 512Mb / 256Mb / 128 Mb in one chunk of memory was added. This is also needed to map memory 1:1.

- Configure driver domain's kernel. Typically, this step consists of a proper kernel configuration and turning-on of hardware drivers, which will be used in a current driver domain.

- Prepare the device tree. Device tree for a driver domain is still attached to kernel due to unresolved passthrough issues, which were described in the following article: http://wiki.xenproject.org/wiki/File:Device_passthrough_xen.pdf. Having analyzed all the possible solutions we decided to keep the driver domain device tree blob attached to kernel. In general, this is not the best solution. One of the proper solutions is to have only one device tree for a trusted domain 0, and control logic in xentools, which copies "passthrough" nodes as is to a driver domain on the fly during its creation.

- Bring up PV drivers' backends. As soon as the driver domain controls hardware, PV backends migrate here. On Nautilus we have PV audio, PV framebuffer, PV event device, and PV USB backends, which we moved from a domain 0 to a driver domain.

After these steps are completed, the driver domain is ready to run.

Hardware distribution conflicts between driver domains

Resolution of such conflicts is one of the steps of a driver domain bring-up. I decided to describe it in separate chapter because it is the most risky one. Depending on the requirements you may succeed in hardware distribution or not. For example, a hardware audio driver is needed in one driver domain and a hardware video camera has to be operating in another driver domain. And now imagine that they both use the

same I2C controller or DMA controller. Thus, it is needed to share the same I2C / DMA cores between domains. But how can it be done without a lot of ugly tricks in I2C / DMA driver?

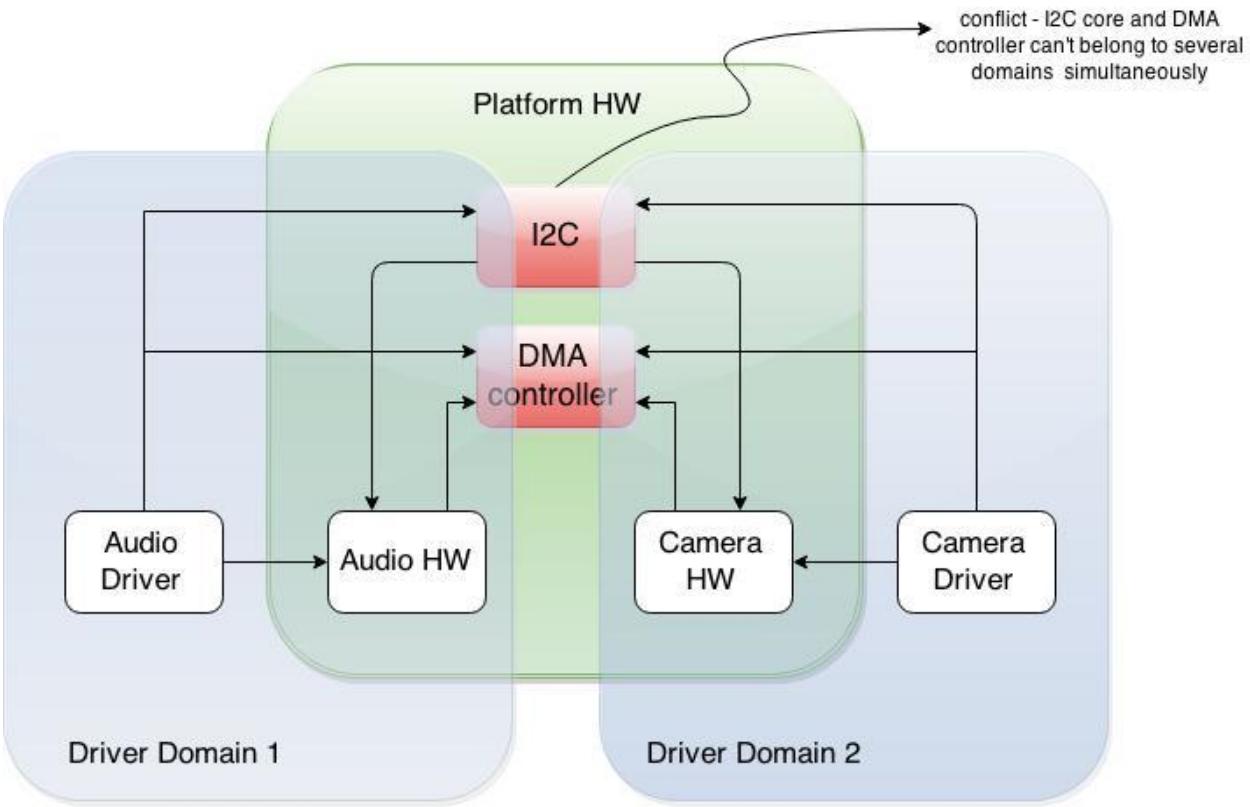

Figure 1. Conflict in IP cores distribution

Looking deeply, you may be lucky on the IP cores level, namely your audio and camera may use different I2Cs, different DMAs, different GPIOs, but still they may use the same clock oscillator and the same voltage controller. And in case one driver touches them, you cannot even predict the behavior of another driver domain. **This is the most risky issue, as it leads to a completely unpredictable behaviour and impacts the whole system's stability**.

Here are some real-life examples:

- I2C IP crashes unpredictably if its parent clock oscillator is scaled to an initial value by a clock framework of another driver domain.

- DMA controller returns unpredictable errors in case its parent clock oscillator is scaled to an initial value by a clock framework of another driver domain.

- Things are even worse if one of the driver domains uses aggressive power management. In this case, if IP Core turns idle its parent clock oscillator turns off

in one driver domain, and at the same time another driver domain fails during access to a disabled hardware, because the same clock is a parent of another IP core, which belongs explicitly to this domain.

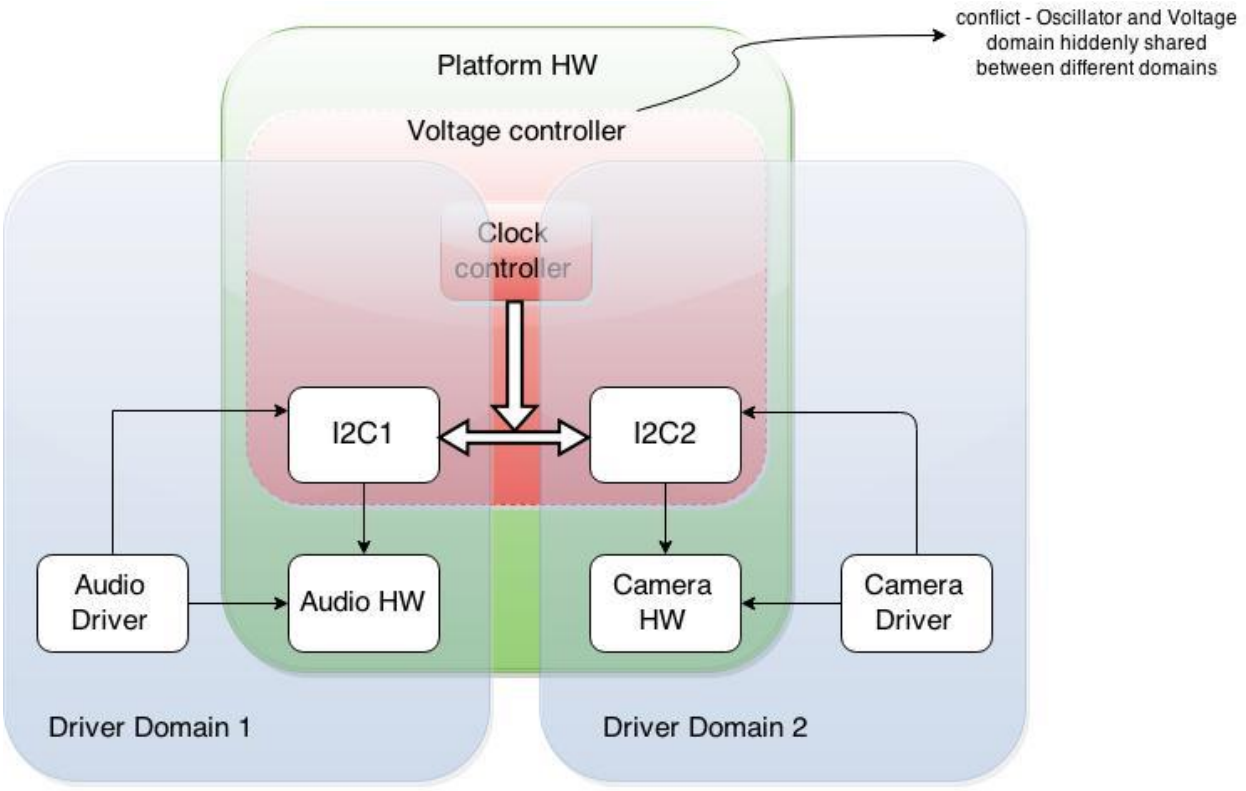

Figure 2. Conflict in hidden distribution of IP cores

The possible solutions are as follows:

- Do not touch shared HW resources in several domains. This is true for such examples as one clock and one voltage controller for several IPs. For example, these resources may be configured once they are in a bootloader, or once they are in a driver domain, whichever starts first. This solution requires a deep low level understanding of resources communication inside a specified platform, as it is hard to say which resource cannot be touched in a specified domain.

- Distribute everything, including clocks and voltage domains. This will be described in details in the following chapter.

Complete hardware distribution in virtualized environment

We are lucky with our Nautilus system. Having a deep understanding of the platform's low level internals we succeed to distribute IP cores without serious conflicts. In this chapter I want to describe how to get rid of such conflicts completely. The idea is quite

simple: you need to redesign the connection of hardware elements taking into account their distribution between domains. A part of this job can be done with existing hardware. For example, some clock oscillators allow reconnection to different parents. With this technique it is possible to minimize the impact of hardware sharing. But the best idea is a complete HW distribution during platform hardware's design. This can be done if a platform is designed for a specialized market (automotive, in our case) with a view that a platform will work under virtualization.

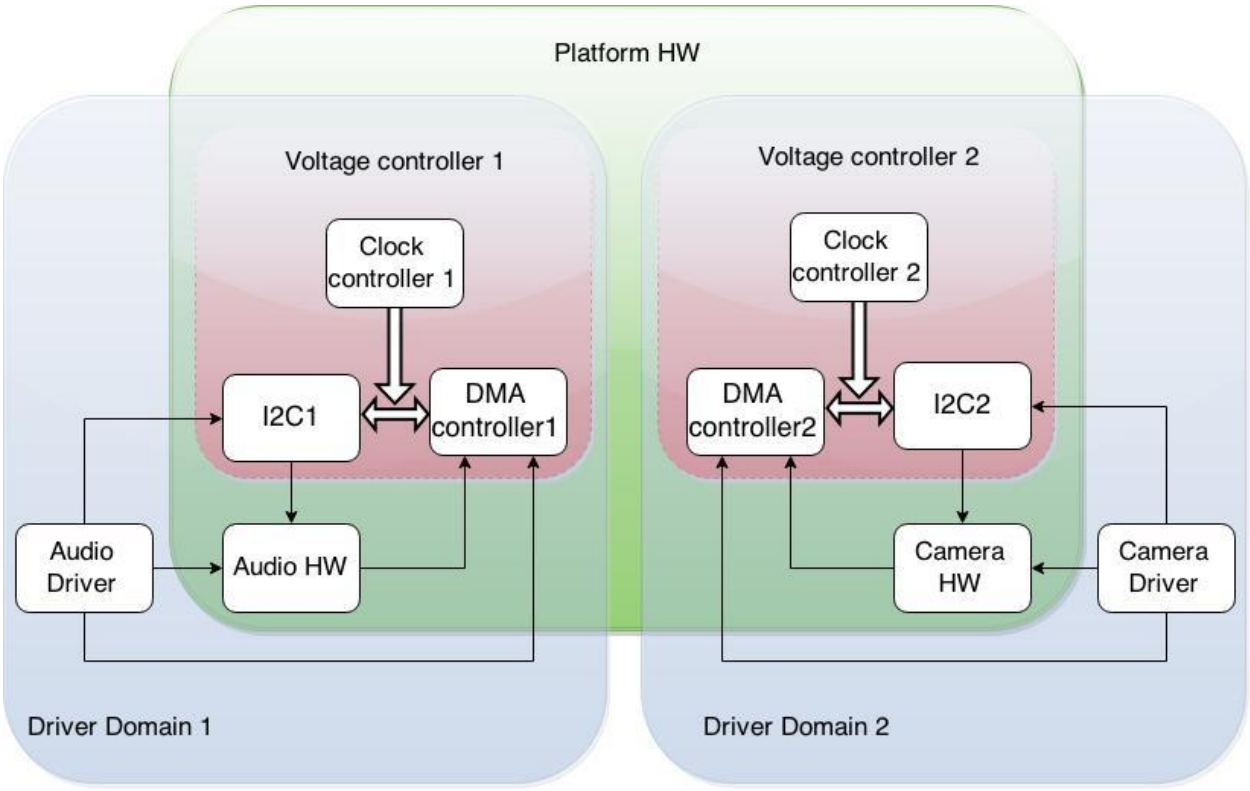

Figure 3. Complete hardware distribution between driver domains

In this case the following conditions should be met (the list is non-exhaustive):

- There should be as many external clock oscillators as there are domains in your system. Note that system clocks are roots of other clocks of a platform. All domains should be counted, not only driver domains.

- There should be as many I2C buses (or any other control buses) as there are domains in your system and they must be powered by clocks inherited from an explicit domain system clock.

- There should be as many GPIO controllers as there are domains in your system, with the same requirements as for I2C buses.

- There should be as many DMA controllers as there are domains in your system, with the same requirements as for I2C buses.

- In case of power management usage, voltage regulators must be taken into account. One driver domain may use several voltage regulators, but they must be specifically allocated for it. No other domain should access them.

Of course, this is an ideal situation. So far, I am not familiar with ARM-based platforms which follow these requirements. I hope that the next generation of well-known automotive vendors will follow this principles. This won't seriously impact the platform's price, and the one who implements it will definitely be a market winner.

Driver domain crash and reboot

Finally, the main advantage of a driver domain is that it can crash and system will remain alive. On Nautilus platform we guarantee that even if a driver domain (or any other domain) crashes, the system will stay alive together with all critical automotive stuff. You only need needed to restart it and reconnect PV backends with corresponding PV frontends. For example, if a driver domain crashes for some reason while audio is playing under Android, for an end user it will look just like a pause in audio playback. The pause will take long as it is needed for kernel + PV audio backend restart.

And what about domain 0?

Yet, after all hardware migrated to a driver domain, domain 0 become "thin". Its kernel, together with a device tree is almost empty and contains only SATA driver. The only PV backend which is still running in domain 0 is a PV block-device. As I have mentioned before, for us it means stability improvement. Domain 0 does nothing, except hypervisor-related stuff: domains' creation and management. The only hardware whose stability must be guaranteed is SATA, but it is much easier to guarantee SATA's stability only than a stability of whole hardware platform.

Conclusion

In this article I have briefly described the experience of bringing up a driver domain on an ARM-based platform. I need to say that it is quite easy to do it under a Xen-powered environment, which provides a possibility to introduce such features easily along with a great community support. And it seems to be one of the first driver domains running successfully on an ARM-based platform. At least I haven't found any information that someone does similar work on ARM.