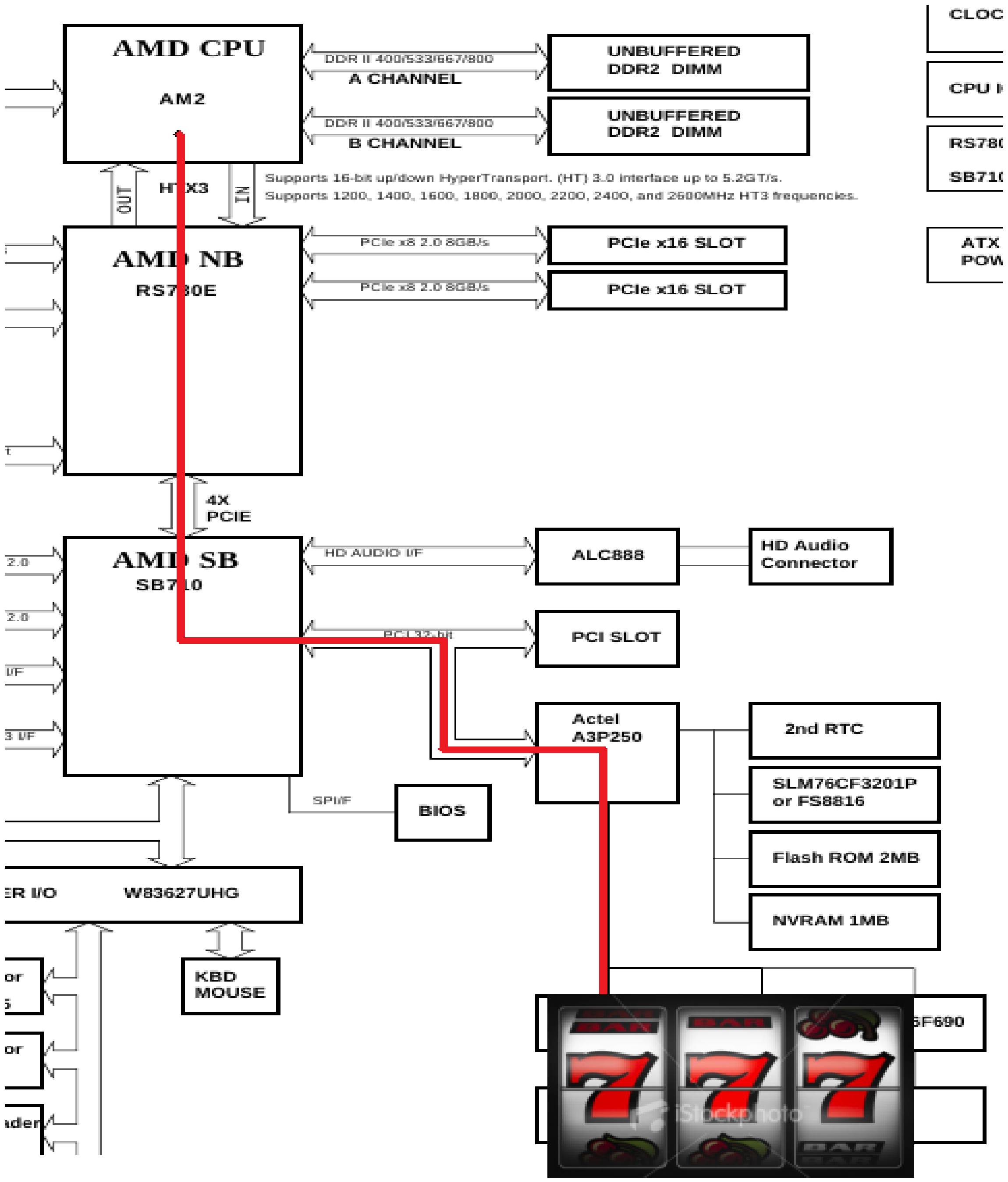
Report on Linux driver for an FPGA driving a stepper motor via PCI bus.

Problem Statement:

The stepper motor is controlled by the FPGA via i/o ports. The application programmer needs control of the stepper motor for developing a game. This problem can be solved by giving an API for the application programmer to use to control the stepper motor. The FPGA is connected to the CPU via an AMD SB710 south-bridge. The south bridge connects to the FPGA via a PCI bus. A driver needs to written which can give specific the PCI slot and thus communicate with the FPGA and finally drive the motor.



The block diagram of the system

The driver in Linux is in the form of a loadable kernel module which can be attached to the kernel at runtime. The task at hand is to read the specifications of the FPGA PCI communication and give specific commands to the PCI slot via the driver to drive the motor.

The Motor which is to be controlled via the driver

Understanding the PCI bus

lspci is a good way to see which devices are connected to your system via the PCI bus

```
root@user:~# lspci | grep actel -i

04:05.0 Non-VGA unclassified device: Actel Device 1770 (rev 30)
```

The Actel device connected to the CPU can be seen here.

Each PCI peripheral is identified by a domain number(16 bits),a bus number(8 bits), a *device* number(5 bits), and a *function* number(3 bits). Making up a 32 bit address. This address is visible in the lspci output. Same information can be obtained from listing all directories of /sys/bus/pci/devices the addresses of each devices can clearly be seen here.

```
root@user:~# ls /sys/bus/pci/devices

0000:00:00.0  0000:00:05.0  0000:00:12.1  0000:00:13.1  0000:00:14.1  0000:00:14.4
0000:00:18.1  0000:01:05.0  0000:00:01.0  0000:00:11.0  0000:00:12.2  0000:00:13.2
0000:00:14.2  0000:00:14.5  0000:00:18.2  0000:02:00.0 0000:00:04.0  0000:00:12.0  0000:00:13.0
0000:00:14.0  0000:00:14.3  0000:00:18.0  0000:00:18.3  0000:03:00.00000:04:05.0
```

The Driver

While writing this device driver three things were kept in mind

- It is a character device
- It is a PCI device
- The I/O is memory mapped

Memory mapped I/O operation is used in this PCI driver because of the following advantages it has over I/O ports

- it doesn't require the use of special-purpose processor instructions
- CPU cores access memory much more efficiently, and the compiler has much more freedom in register allocation and addressing-mode selection when accessing memory.

The PCI hardware responds to three types of queries:
Memory Locations
I/O ports
Configuration Registers

Accessing the configuration registers of the device will enable the driver to know the memory addresses the I/O is mapped.

The driver is registered in the kernel by the *struct pci_driver*.

The driver needs to maintain a table of devices it supports and it does that via maintaining the *struct pci_device_id*.

When a device of having an ID in the given table is found it is known that the driver in the *pci_driver* will support it.

Each driver has to have an init function and an exit function. This driver also has a probe function which is called when the kernel has found a device for the driver registered by the init function.

INIT function
The function of the init function is to register the driver and also since we are writing a driver a character device we also need to obtain device numbers of character devices we are handling. This is accomplished by *alloc_chrdev_region* which dynamically a device number in case we do not want to specify a specific major number

PROBE function
Before describing about the probe function it should be noted that it is a convention to maintain a struct storing all the device related information.
This would contain the following items:

- -struct pci_dev i.e. the device specific struct for the pci device maintained by the kernel
- -the physical memory addresses assigned to the pci device
- -the kernel virtual memory addresses assigned to the device
- -device number struct(char devices)
- -struct cdev(char devices)
- -class information

The probe function has to do the following tasks in the same order:

- -Set up char device by registering it with kernel. Done by *cdev_init, cdev_add* functions
- -Enable or wakeup the pci device done by *pci_enable_device*
- *-A*ccess the configuration registers to know the physical address of the pci device. Done by wrapper functions like *pci_resource_start.*
- -After knowing the physical addresses we request these regions to be allocated to the particular device. Done by function *request_mem_region.*
- -Then to make these addresses accessible to the kernel we need to map these to kernel virtual addresses done by *ioremap* function
- -Create a sysfs entry for the device. This is accomplished by the *device_function* function

note: The fpga specification sheet states that the bar 3 of the pci device has been used to drive the motor. Hence while accessing the configuration registers of the pci device, the physical address of bar 3 are only obtained. If more bars are used or we do not know which bar is used there should a loop to know physical addresses of all the bars.

REMOVE function
This function is called when the device is removed from the PC'c hardware slot. It releases the corresponding memories and unregisters the device.

EXIT function
This function is called when kernel module is removed. It destroys device, and frees all the allocated memory.

IOCTL function

For the operation of the character device wee need *struct file_operations* to be defined. This maps the functions to be called for i/o operation on the device. This struct is used here to map the ioctl function defined in the driver to the system call for ioctl on the corresponding file in "/dev/fpga" for example.

The IOCTL function is nothing but a switch case statement calling ioread32 and iowrite8 from the proper memory addresses of the virtual memory.
The IOCTL function has a void pointer as an argument for exchange of data between user and kernel space. The data pointed by the void pointer can be accessed only when it is copied from the user space to kernel space. Direct dereferencing the pointer will not work. Same is the case with writing data to user space. Simply passing a pointer would not work. copy_to_user function has to be called.

The data to be exchanged via the void pointer is the offset to the address

One interesting feature of ioctl commands: the commands need to be unique in order to avoid clashes with other devices. So if an ioctl is called with the wrong device it does result in unexpected action but just returns with an -EINVAL. For these a mechanism of a magic number is to be used. All magic umbers are listed in the documentation for linux kernel. Hence I have used a magic number 'y' with sequence nos. 32-33 to avoid collision with any existing device. Macros have been defined in ioctl.h which help to generate unique cmd arguments using magic numbers and sequence numbers.

Using the driver

The usage of this driver is a call to open on "/dev/fpga" followed by the required calls to ioctl.

Video

This is a video of the working test program using the driver written here.
http://www.youtube.com/watch?v=1QdxbfY0dSk

Miscellaneous learnings

The inode struct has a pointer to a character device if the corresponding inode is used for the character device. This pointer can be used to access the character device from the inode.

printk() with different levels of kernel priority messages eg. KERN_INFO,KERN_ERR,KERN_ALERT were useful in debugging while writing the device driver.

The LXR linux cross reference proved to be very useful to browse the source code of the kernel.

The book www.makelinux.com/ldd3 proved to be very useful in understanding the key concepts of the linux device driver writing.

References

- Chapters 3,6,8,9 and 12 from Linux Device Drivers 3rd ed. www.makelinux.com/ldd3
- Documentation in the Linux kernel source regarding the PCI drivers http://lxr.linux.no/linux+v2.6.38/Documentation/PCI/
- Source code of PCI drivers in Linux kernel http://lxr.linux.no/linux+v2.6.38/drivers/pci/
- Source code of Linux driver for PCI implemented on Altera http://marc.info/?l=linux-kernel&m=122813921631142&w=2
- Source code for Linux device driver for PCI enabled FPGA provided by AEWIN.
- Essential Linux Device Drivers by Sreekrishnan Venkateshwaran. Print ISBN-10: 0-13-239655-6,Print ISBN-13: 978-0-13-239655-4