14 July 2015

In this tutorial two different approaches to accessing FPGA resources from application will be shown. The assumption is that the FPGA resources are memory mapped registers.

For this tutorial, we simply installed two registers as shown in table below

	hps_0.h2f_axi_master	hps_0.h2f_lw_axi_master
sysid_qsys.control_slave		0x0001_0000 - 0x0001_0007
hps_0.f2h_sdram0_data		
hps_0.f2h_axi_slave		
FPGA_preloader_memory.s1	0x0000_0000 - 0x0000_ffff	
UltiEVC_AX_0.avalon_slave		0x0000_0000 - 0x0000_ffff
UltiSDC_0.avalon_slave		0x0003_0000 - 0x0003_ffff
pio_OUT.s1		0x0001_0020 - 0x0001_002f
pio_IN.s1		0x0001_0010 - 0x0001_001f

We will write a value in `pio_OUT.s1` and readback value from `pio_IN.s1`

Approach n.1: accessing resources using the /dev/mem file

As already stated, peripherals are connected to the lightweight HPS-to-FPGA bridge. The lightweight bridge's region of memory begins at address `0xFF200000`, so to find the address of an FPGA peripheral, simply add the peripheral's offset as shown by Qsys to that address. In our case, the `pio_OUT.s1` peripheral was assigned the offset `0x00010010`, so the full address is simply `0xFF210010`. In the same way, the full address of `pio_IN.s1` is `0xFF210020`

The Linux kernel uses virtual memory, so we cannot directly write to address `0xFF200000` from a userspace process, since that physical address is not mapped into the process's address space. Despite the best approach is to write a kernel driver, we will use a simpler method, which is to use the `mmap` system call on the "`/dev/mem`" device file, which represents physical memory, to map the HPS-to-FPGA bridge's memory into the process memory.

The full source code of the application is attached to the post.Ignoring all of the error-handling and setup code, the important parts of the program are the following.

```
gpio_map = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED,
                        fd, gpio_base);
gpio_mem = (unsigned char *)gpio_map;
*(gpio_mem+EXOR_GPIO_OUT_OFFSET) = value;
rvalue = *(gpio_mem+EXOR_GPIO_IN_OFFSET);
```

The `mmap` call maps a single page of memory beginning at `0xFF210000` into the process's memory space. The first argument to `mmap` is the virtual memory address we want the mapped memory to start at. By leaving it `NULL`, we allow the kernel to use the next memory address available. The second argument is

the size of the region we want mapped. The size will always be a multiple of the page size (on Linux, this is 4 kB or 4096 bytes), so we specify the size of a single page even though we only need a byte.

The second line simply cast the `gpio_map` void pointer to an unsigned char pointer.

The, the third line writes to the memory address, setting the value passed on the command line. Notice that `gpio_mem` is declared with the `volatile` keyword. This tells the compiler that the value stored at this memory address can change without being written to from software. This disables certain compiler optimizations that can cause incorrect behavior.

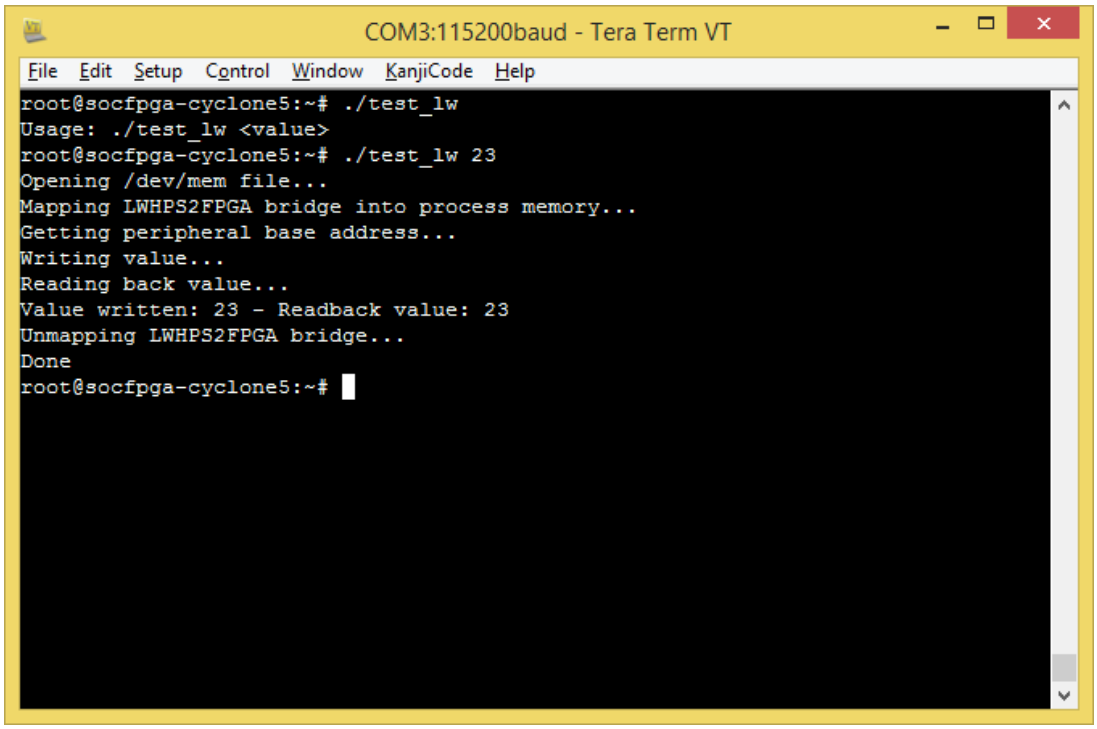
Finally, the fourth line reads back the value

To build the test executable, you will probably have to edit the Makefile to adjust the path to the `gcc` cross compiler then simply type

```
make
```

on the command line. This will create an executable file that you can copy to your target device and run on the target itself

```
scp test_lw root@<target IP address>:/home/root
```



Approach n.2: writing a kernel driver

In this section of the tutorial, we will implement a kernel driver so that the FPGA resources will be accessible through a file in the `/dev` folder. After installing the driver, the value in the FPGA register will be written and read by simply typing

```
echo 1 > /dev/exor_gpio
cat /dev/exor_gpio
```

Back in the old days, a device file was a special file created by running an old shell script named `MAKEDEV` which called the `mknod` command to create every possible file in `/dev`, regardless of whether the associated device driver would ever run on that system. The next iteration, `devfs`, created `/dev` files when they were first accessed, which led to many interesting locking problems and wasteful attempts to open device files to see if the associated device existed. The current version of `/dev` support is called `udev`, since it creates `/dev` links with a userspace program. When kernel modules register devices, they appear in the `sysfs` file system, mounted on `/sys`. A userspace program, `udev`, notices changes in `/sys` and dynamically creates `/dev` entries according to a set of rules usually located in `/etc/udev/`.

As in the previous section, we will set the delay by writing a byte to physical memory at address `0xFF210000`. However, this address is not yet mapped into the kernel's address space, so we will have to that first. Fortunately, the kernel provides functions for properly mapping and accessing the memory space for peripherals, which is termed IO memory.

First, we will need to request exclusive access to the memory region we want to write to.

```
#define LWHPS2FPGA_BRIDGE_BASE 0xFF200000
#define EXOR_GPIO_OFS          0x00010000
#define EXOR_GPIO_BASE         (LWHPS2FPGA_BRIDGE_BASE + EXOR_GPIO_OFS)
#define EXOR_GPIO_SIZE PAGE_SIZE

res = request_mem_region(EXOR_GPIO_BASE, EXOR_GPIO _SIZE, "exor_gpio");
if (res == NULL) {
    /* do some error handling */
}
```

`EXOR_GPIO_BASE` is set to the base address we want to access, and `EXOR_GPIO_SIZE` is set to the page size (it must be a multiple of the kernel page size, which is 4kb). As with the `mmap` system call, we can only get memory a page at a time, so it makes sense to just request a whole page. Now that we know we have exclusive access, we need to map the address into virtual memory.

```
void *exor_gpio_mem;

exor_gpio_mem = ioremap(EXOR_GPIO_BASE, EXOR_GPIO _SIZE);
if (exor_gpio_mem == NULL) {
        /* error handling */
```


```
}
```

We can now write to `exor_gpio_mem` to set FPGA registers. Of course, it's not considered proper to just do `*exor_gpio_mem = value`. Instead, we should use the `iowrite*` functions. In our case, we are writing a single byte, so we use `iowrite32` . All these functions are defined in `include/asm-generic/iomap.h`

In order to integrate the kernel in the kernel building process, some further steps are required

1. Copy source files

The folder containing source files needs to be copied in the drivers folder of the Linux kernel source tree. Source files include a file named `Kconfig` that tells the Linux kernel configuration user interface about the new Exor GPIO driver

2. Modify the Kconfig file in the drivers folder

To make the Exor GPIO driver visible to the Linux Kernel configuration interface, it must be added at the end of the `Kconfig` file that already exists in the drivers folder

The new lines are highlighted in the listing below

```
source "drivers/phy/Kconfig"
source "drivers/exor/Kconfig"
endmenu
```

3. Change Makefile in the drivers folder

Append the highlighted line at the end of the Makefile in the `drivers` folder to make the build process know that the content of the `exor` folder has to be built when the `EXORSYSGPIO` option is selected

```
obj-$(CONFIG_IPACK_BUS)         += ipack/
obj-$(CONFIG_NTB)               += ntb/
obj-$(CONFIG_EXORSYSGPIO)       += exor/
```

The `EXORSYSGPIO` option is defined in the `drivers/exor/Kconfig` file

```
#
# Exor device configuration
#

menu "Exor devices"


config EXORSYSGPIO
        tristate "/dev/exor_gpio virtual device support"
```


```
        default y
        help
          Say Y here if you want to support the /dev/exorgpio device

endmenu
```

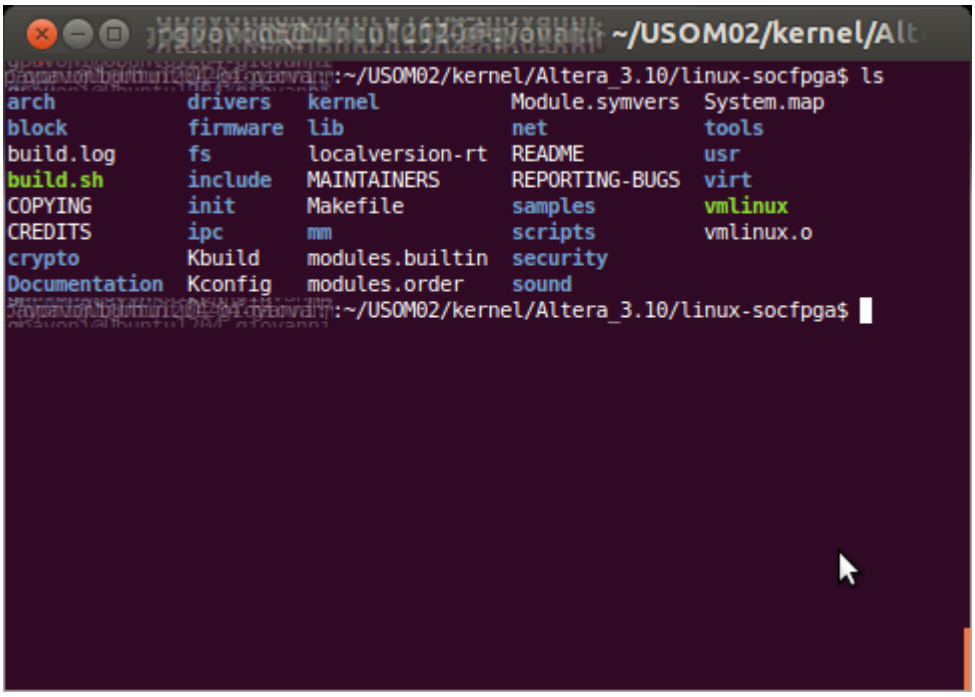
The tristate option make it possible to choose whether to load the module as a static driver (the driver is compiled in the kernel image) or to make it a loadable module that can be loaded using the `insmod` utility

4. Change kernel configuration

Now we need to change kernel configuration in order to

1. enable dynamic module loading and unloading (this make it easy to develop a new kernel driver since you just need to copy a single .ko file instead of the whole zImage)
2. enable the building of the new Exor GPIO driver

In a terminal, enter the base folder of Linux kernel source tree. If you type ls, you should see something like this



Type

```
make ARCH=arm menuconfig
```

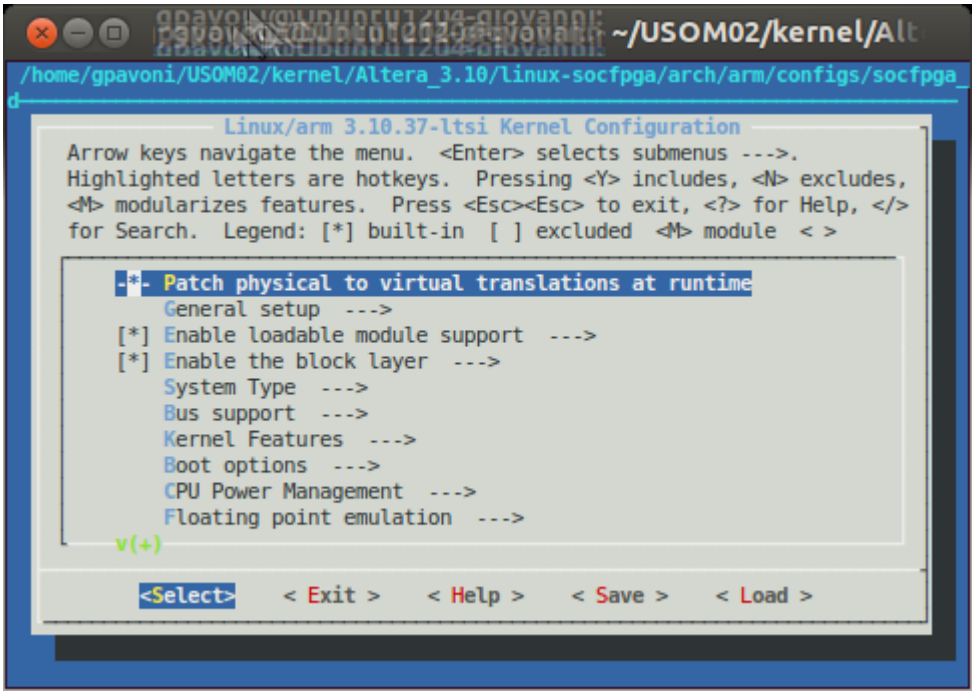
to start the configuration interface. Using the right arrow key, select "<Load>".

14 July 2015

Enter the following configuration filename

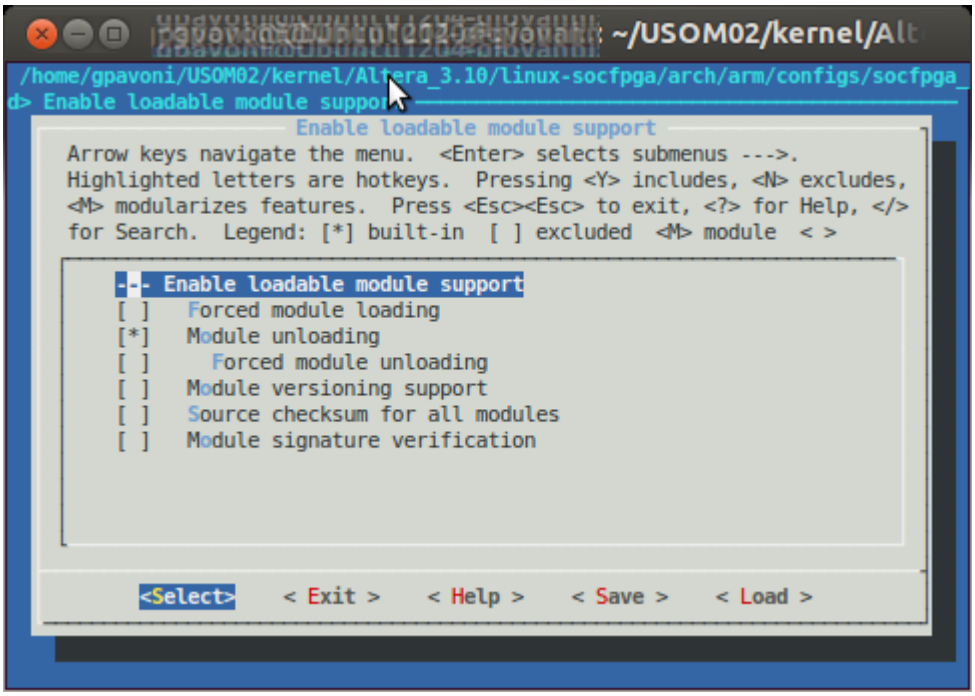
```
<path to the base source tree>/arch/arm/configs/socfpga_defconfig
```

and press `Enter`. You will be bring back to list of configuration options



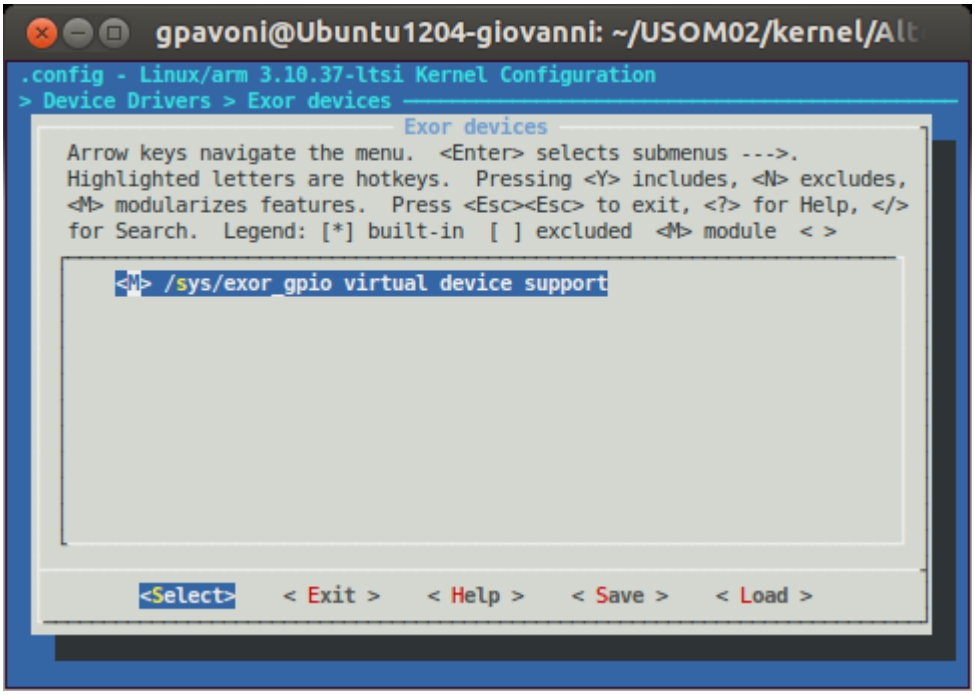
Using down arrow key, select "`Enable loadable module support`" and press `Enter`. In the "`Enable loadable module support`" menu, select the settings as shown in the following screenshot

14 July 2015



Using the right arrow key, select "<Exit>".

Now navigate to "Device Drivers" -> "Exor devices" and select the Exor driver to be a loadable module



Using the right arrow key, select "<Exit>" repeatedly until you reach the first configuration window. Then, again using the right arrow key, select "<Save>" and confirm your intention to save the configuration.

Finally, select "<Exit>" to quit the configuration application. This will bring you back to the console prompt

Now build the kernel by typing the following commands

```
make ARCH=arm socfpga_defconfig
make ARCH=arm CROSS_COMPILE=<path-and prefix of the cross compiler> -j12 zImage
dtbs modules
```

The <path-and prefix of the cross compiler> is the full path to the gcc compiler without the final "gcc". For example, if the gcc compiler's full path is

```
/opt/altera-linux/linaro/gcc-linaro-arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
gcc
```

then the <path-and prefix of the cross compiler> is

```
/opt/altera-linux/linaro/gcc-linaro-arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
```

After the build you should find (all paths are relative to the folder where you started the build process)

- zImage in arch/arm/boot
- socfpga_cyclone5.dtb in arch/arm/boot/dts
- exor_gpio.ko in drivers/exor

zImage and socfpga_cyclone5.dtb need to be copied on the SD card the system boots from. socfpga_cyclone5.dtb have to be renamed as socfpga.dtb.

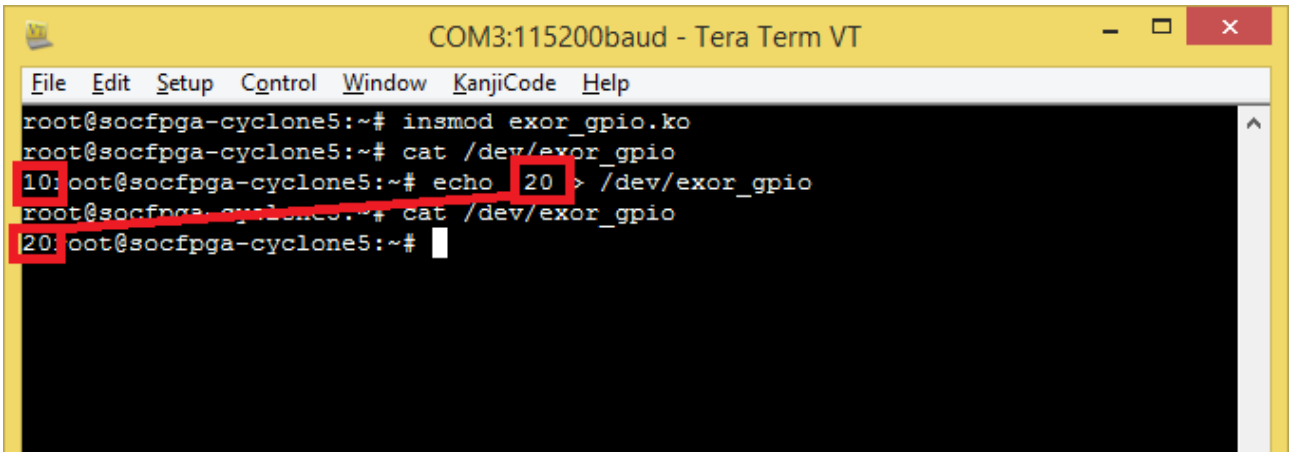
exor_gpio.ko can be copied to any folder of the target system using scp

```
cd drivers/exor
scp exor_gpio.ko root@<target IP address>:/home/root
```

Make an SSH connection to the target and login as root. At the console prompt, you can load the module and test it

```
insmod exor_gpio.ko
cat /dev/exor_gpio
echo 20 > /dev/exor_gpio
cat /dev/exor_gpio
```

14 July 2015



To see the output of the `printk` calls in the driver's source code, type

```
dmesg | tail
```