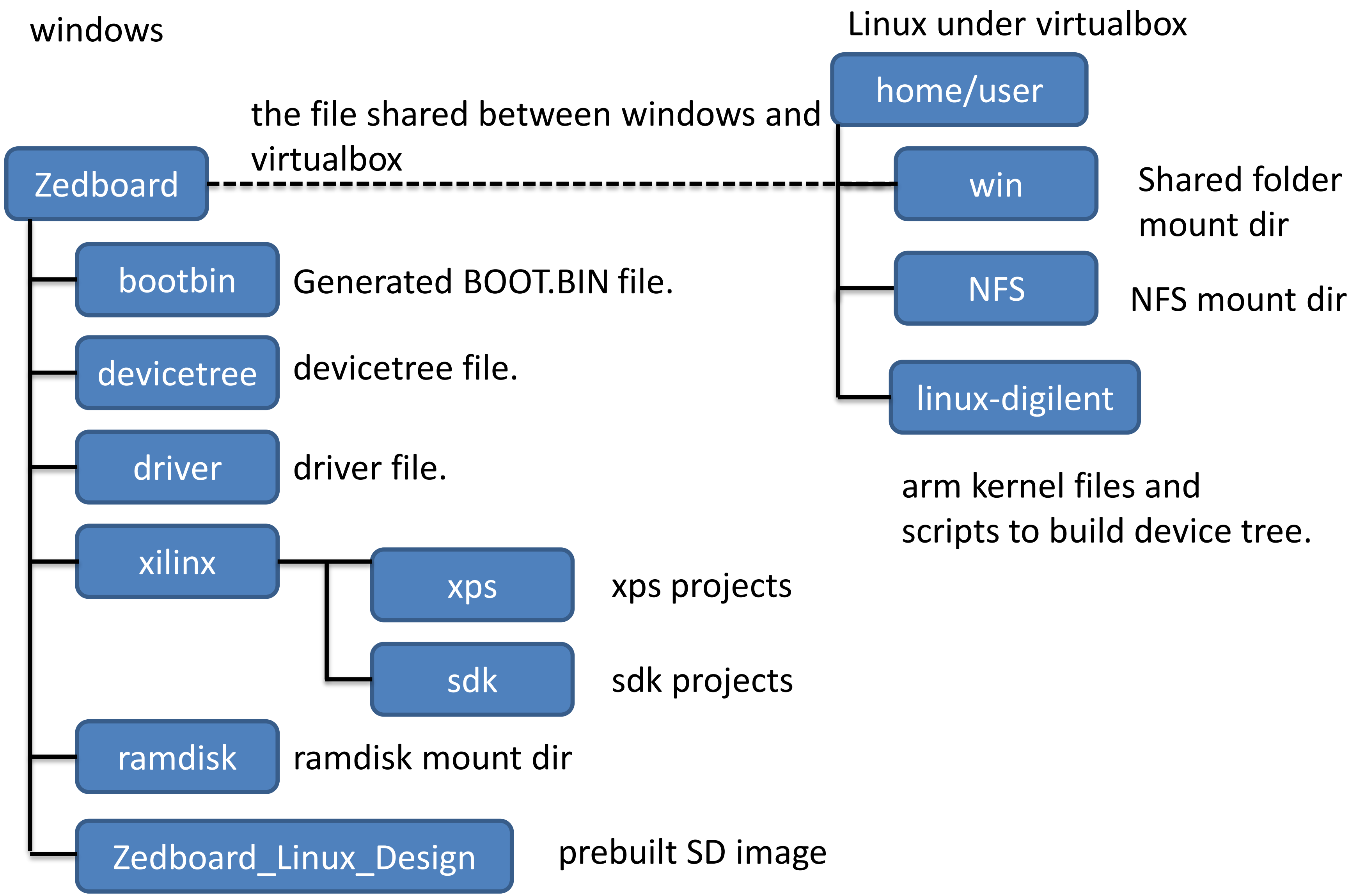
ZedBoard Lab 5
LED and driver

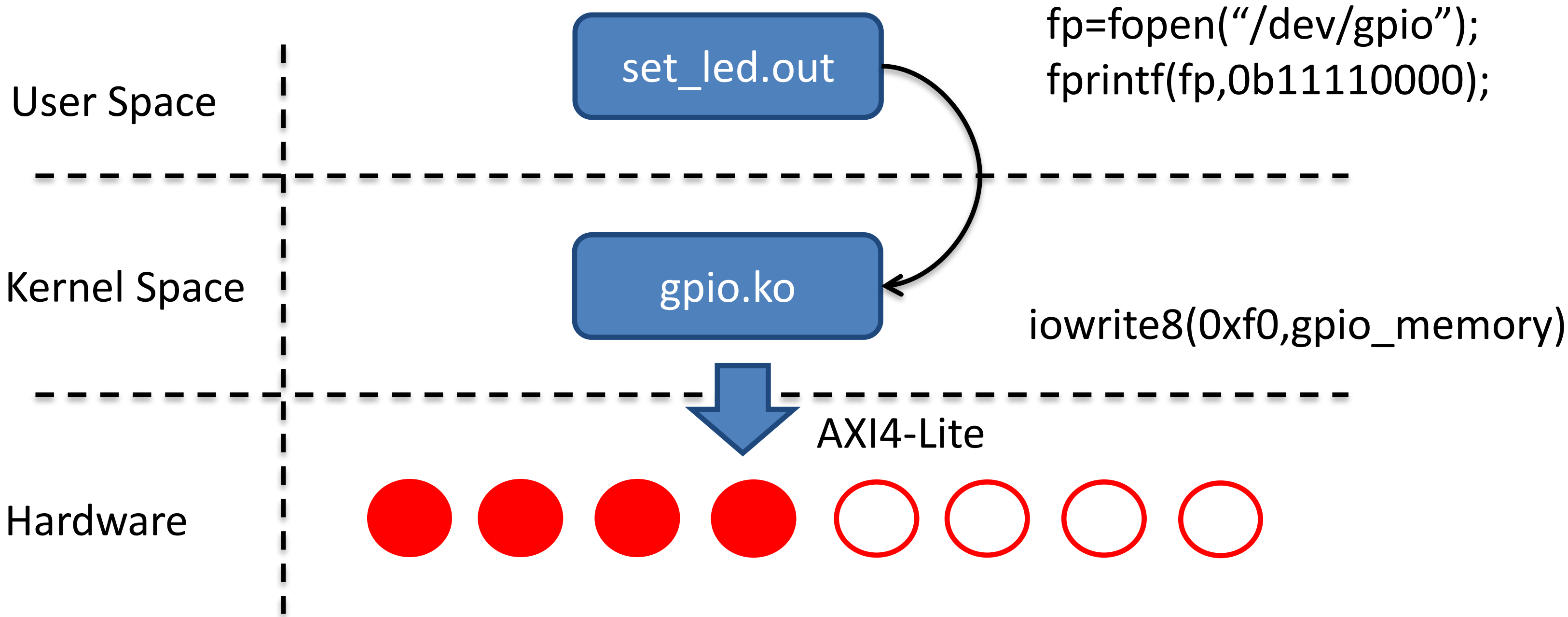
Chun-Chen Tu
timtu@umich.edu

File placement

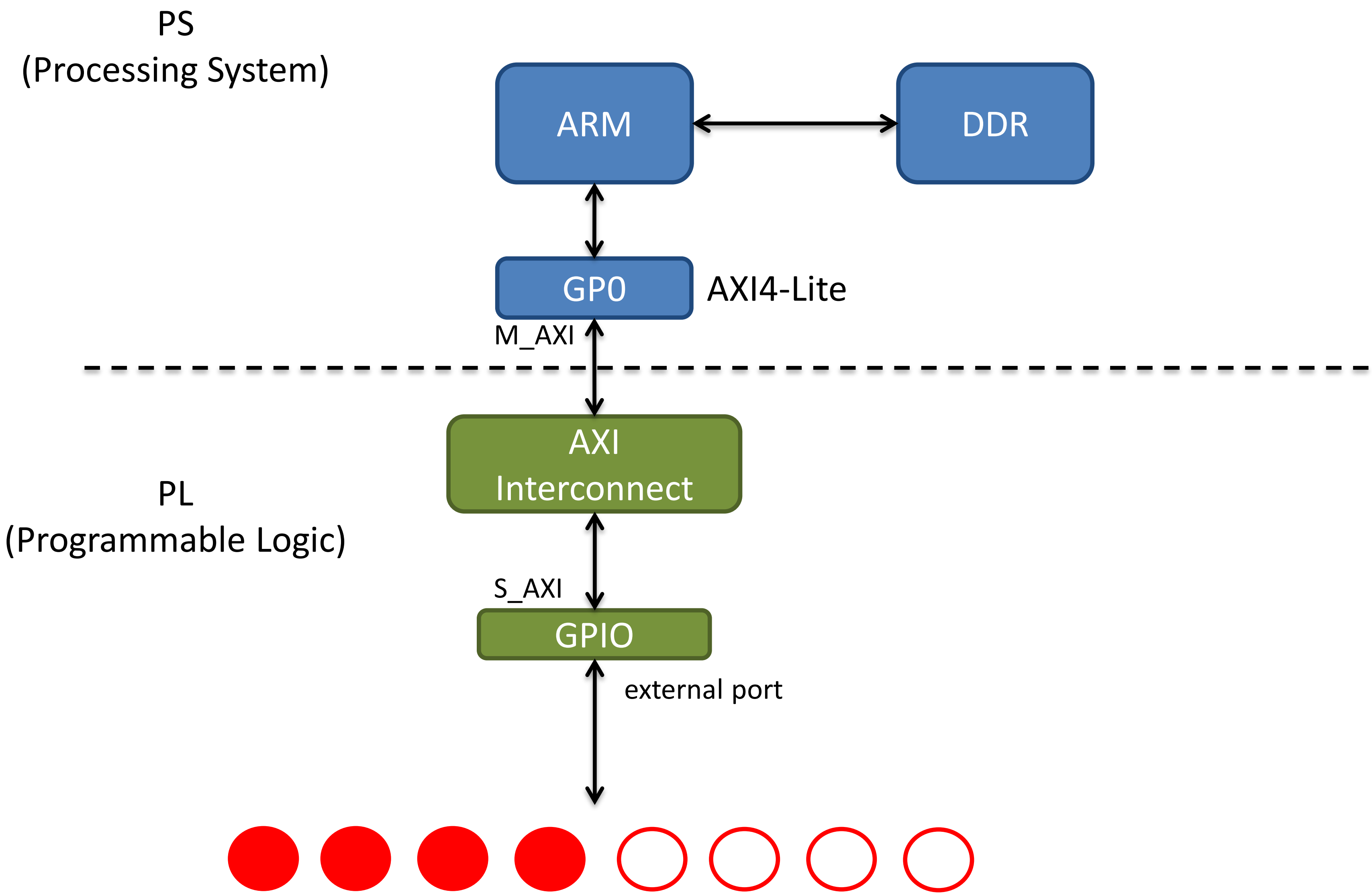


Outline

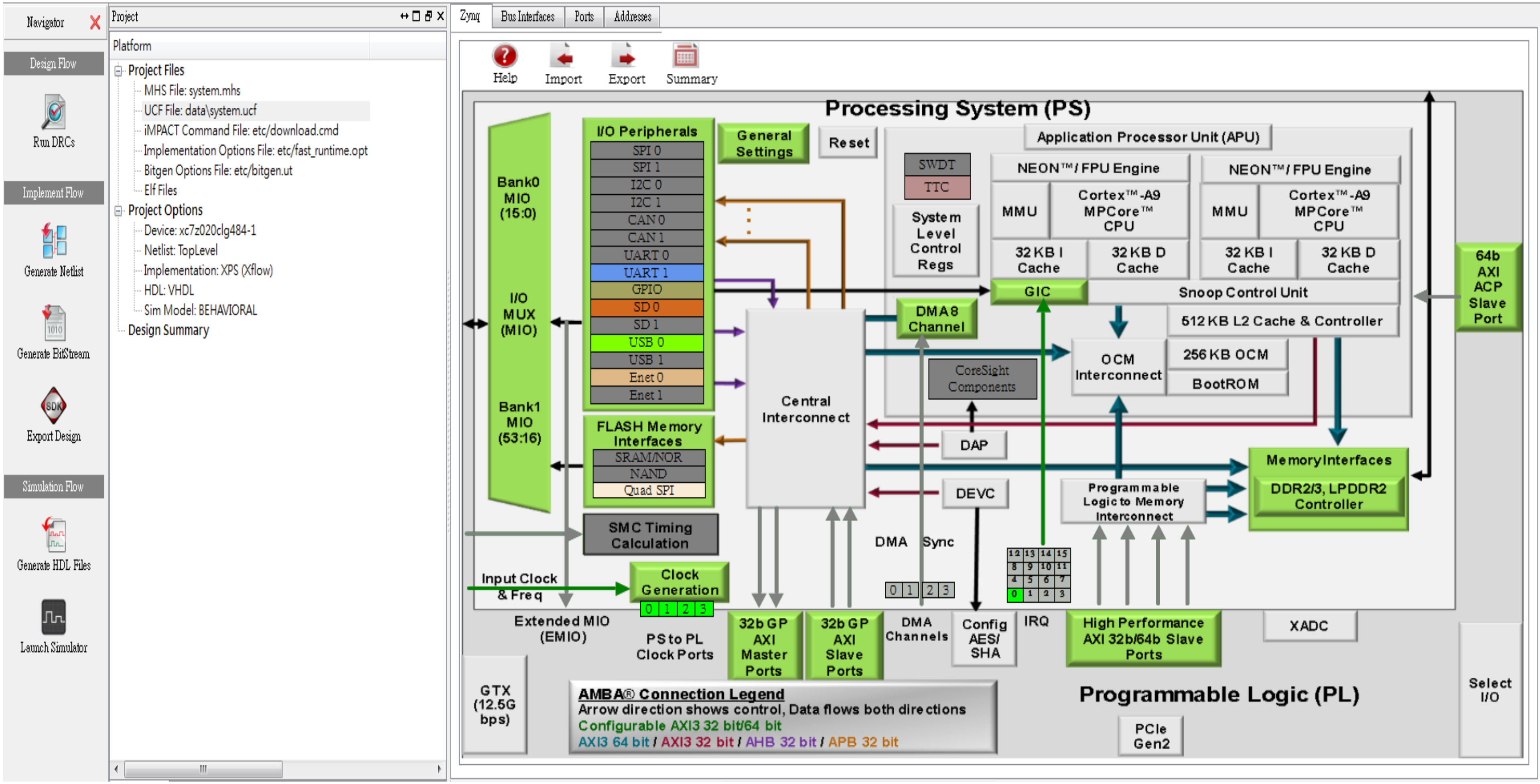
- We’ll add and GPIO (general purpose I/O) to control LED.
- Also, to control LED under OS, we need driver.



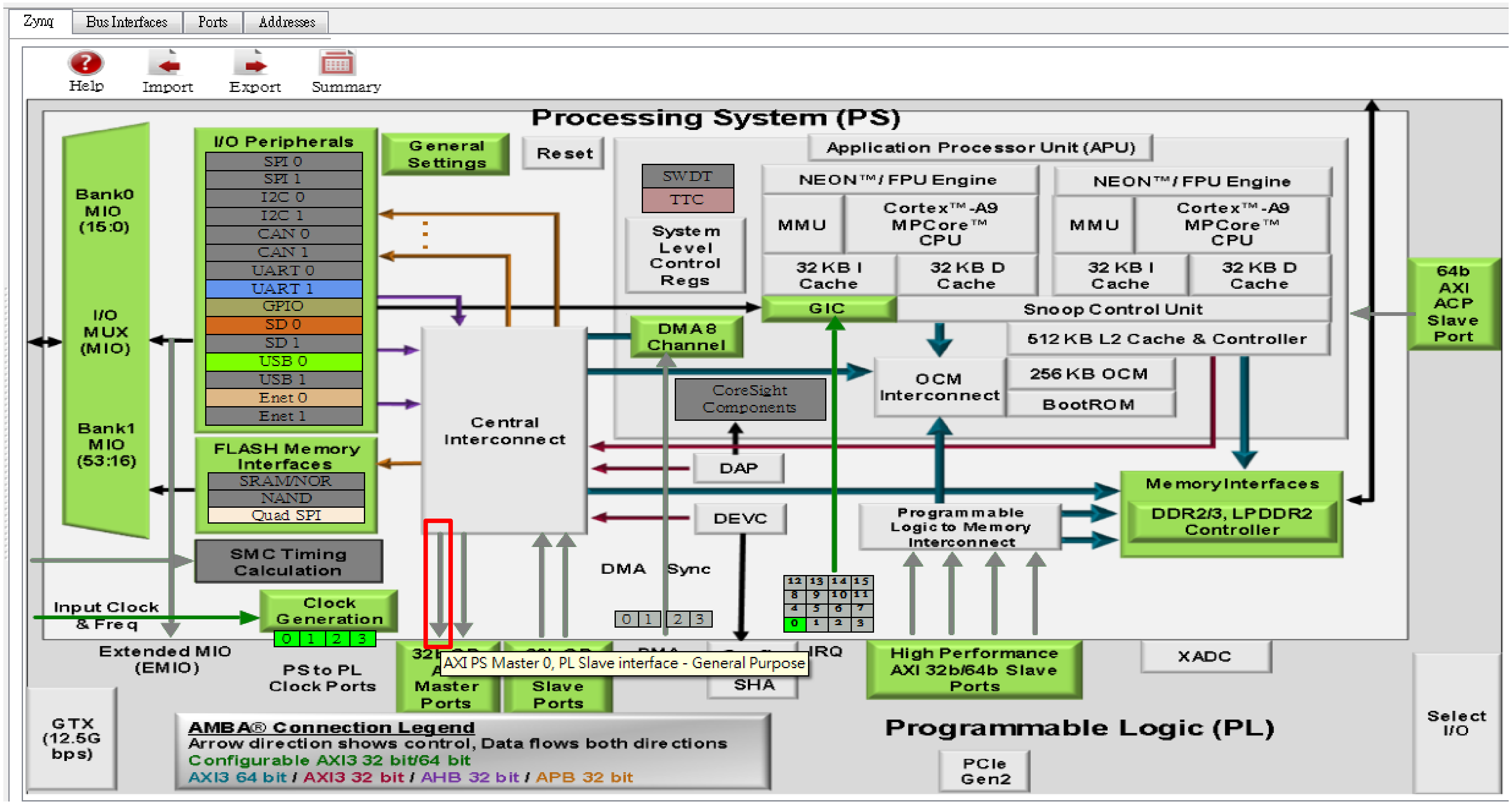
System Architecture



XPS design

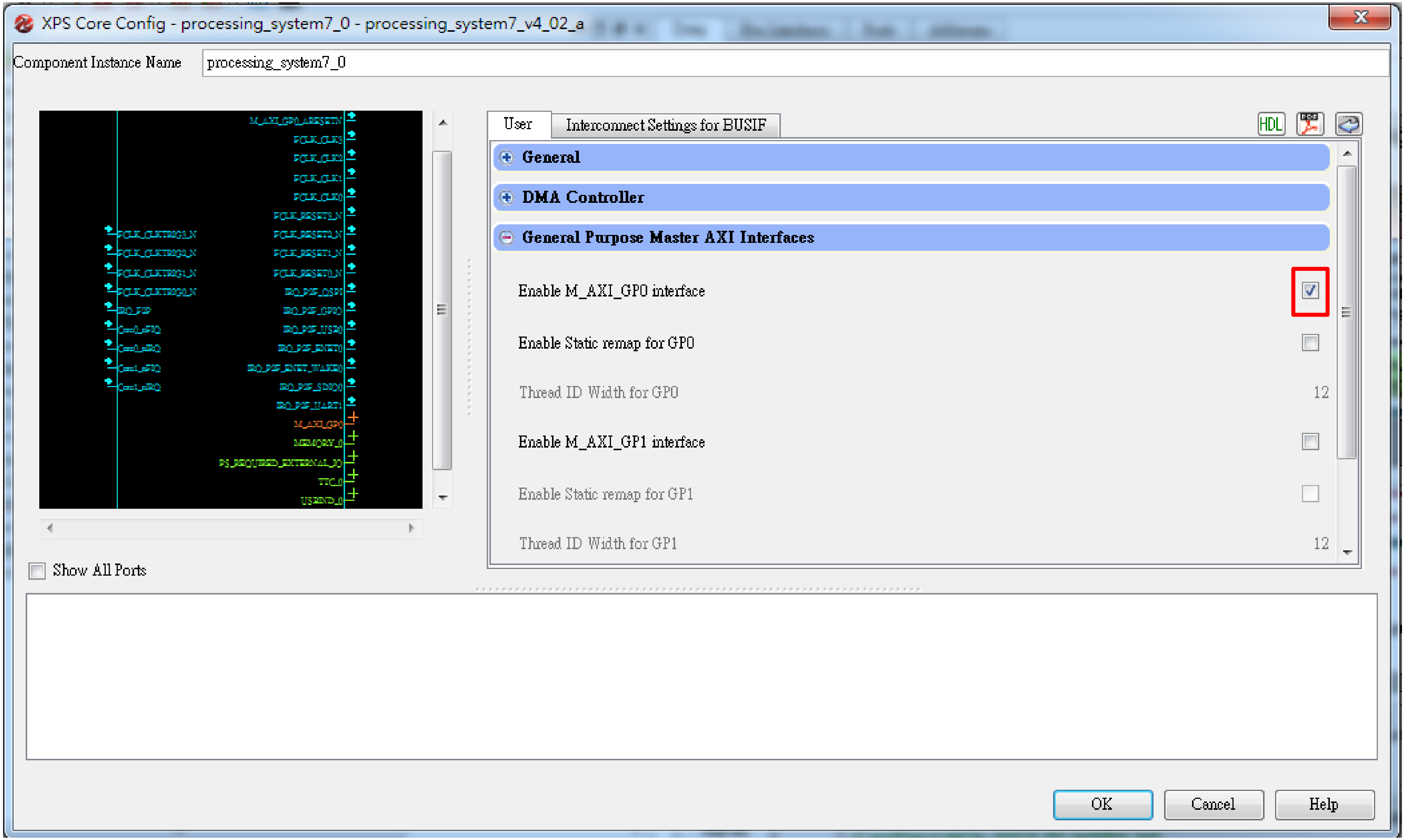


You can create a new project or modify the previous empty design we just created.

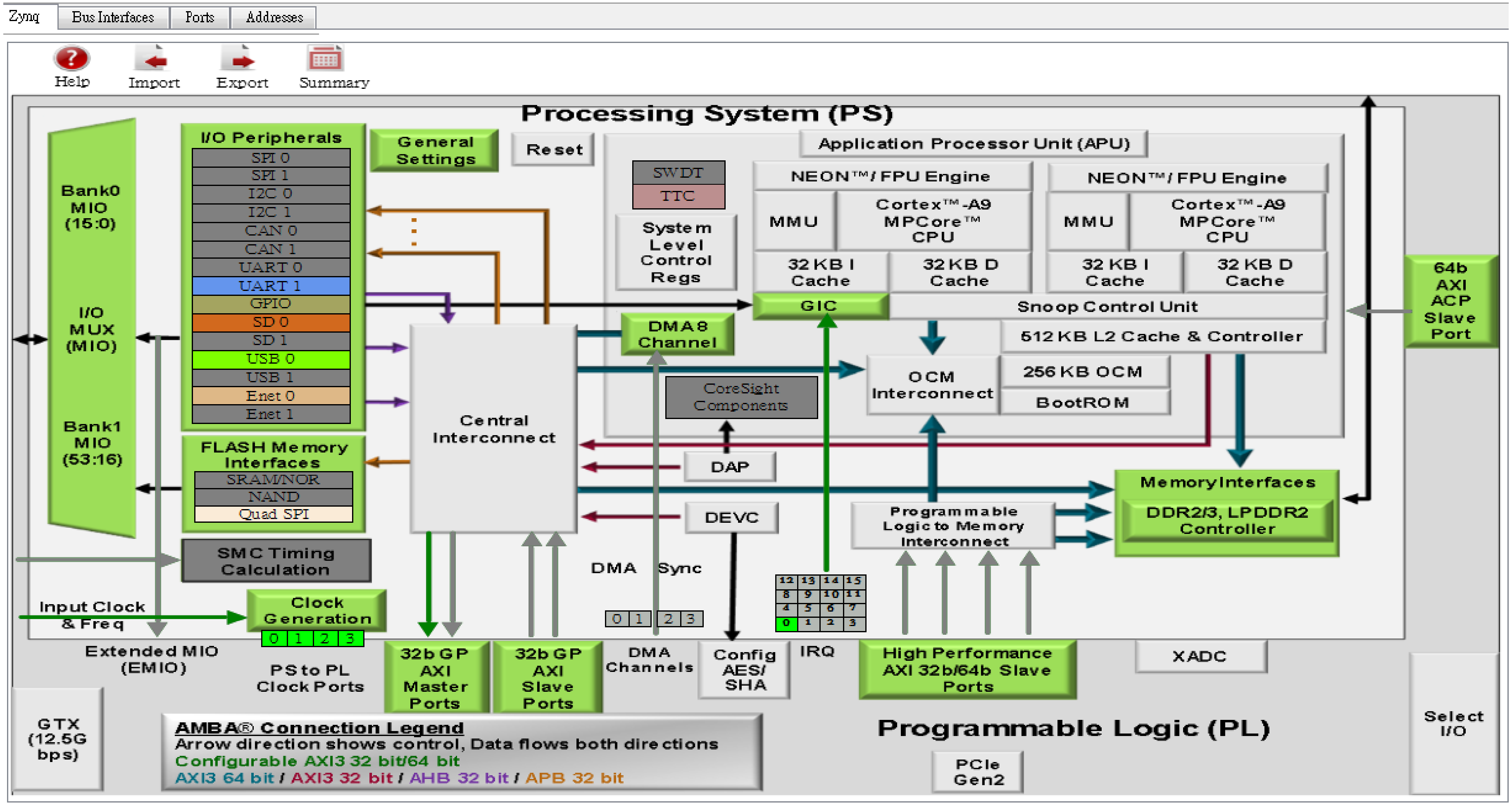


Enable the AXI GP 0 channel.

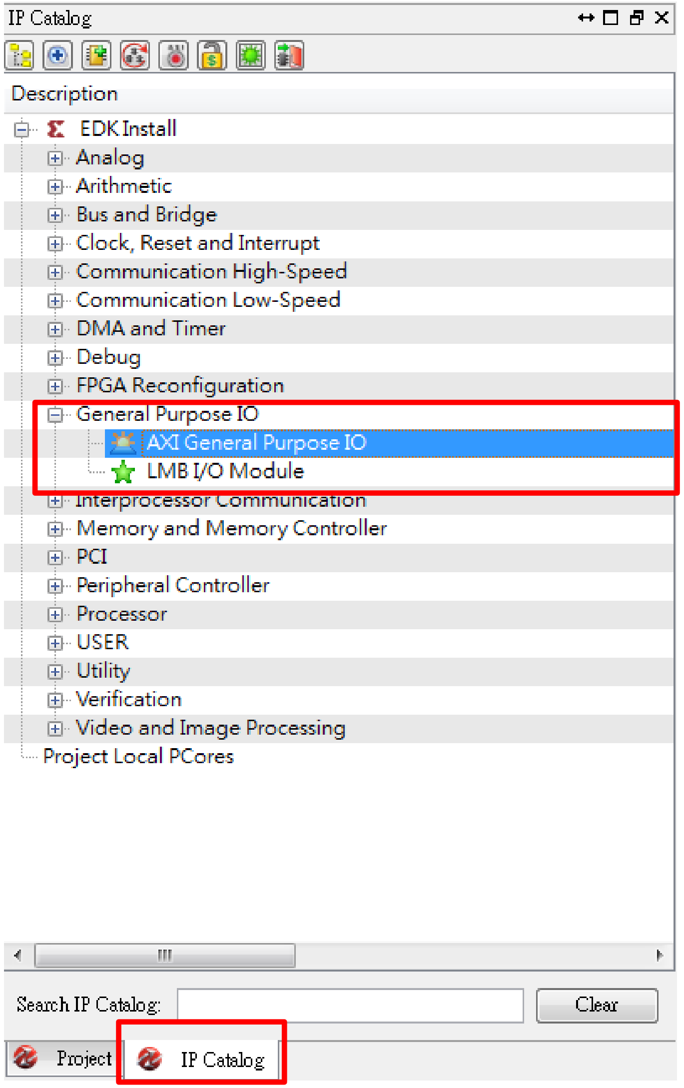
Click on the grey arrow.



Check on the *M_AXI_GP0* box.

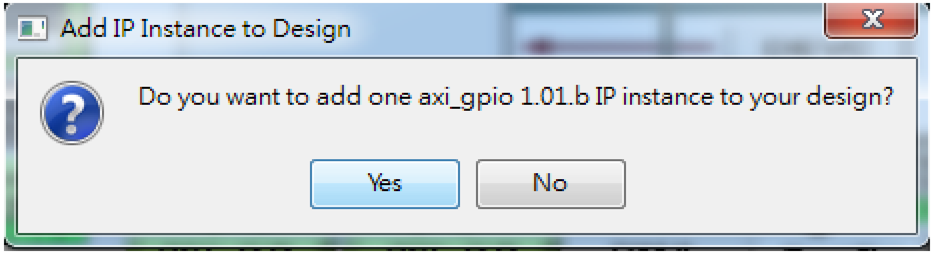


You will see GP0 turns from grey to green. This indicate AXI GP0 is enable.

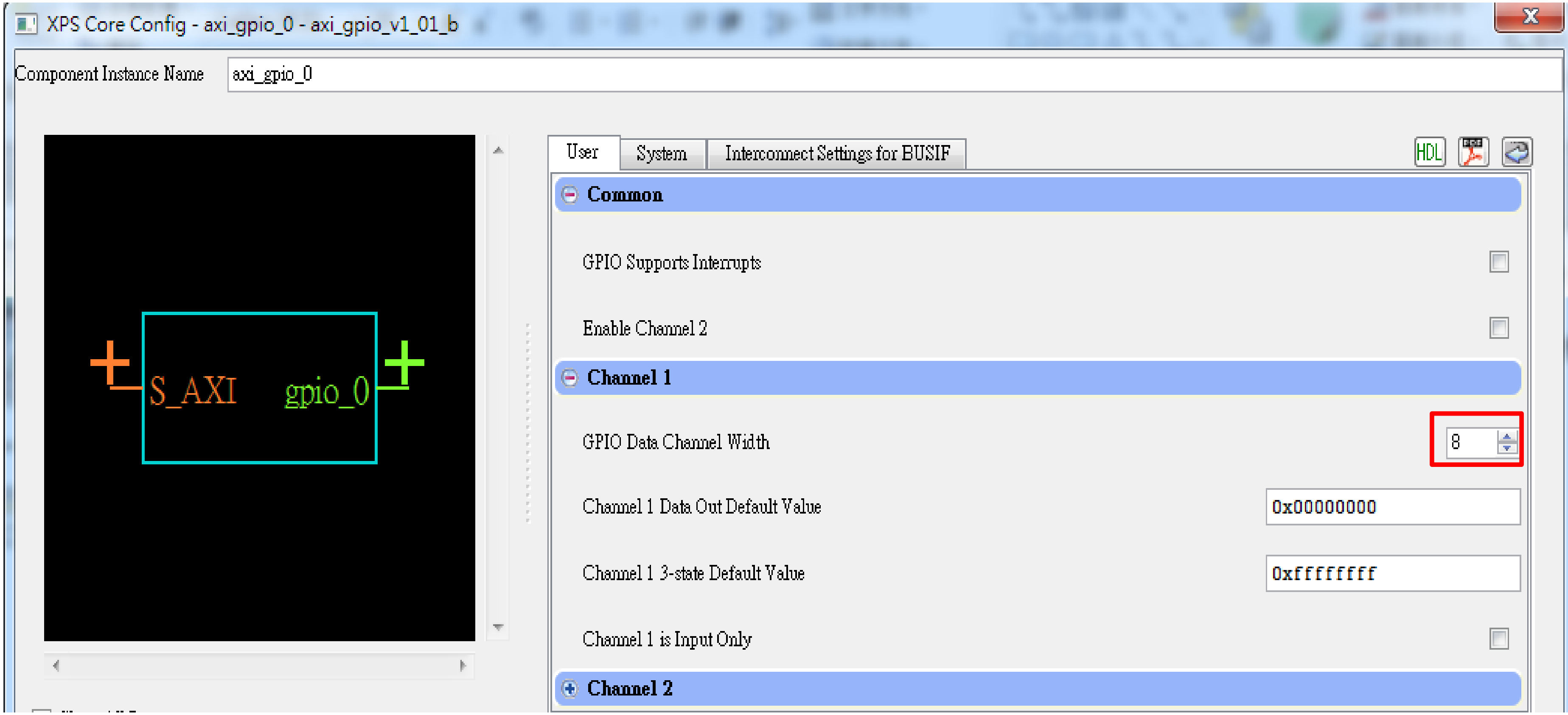


Click on the *IP Catalog*.

Add *AXI General Purpose IO* by double click it

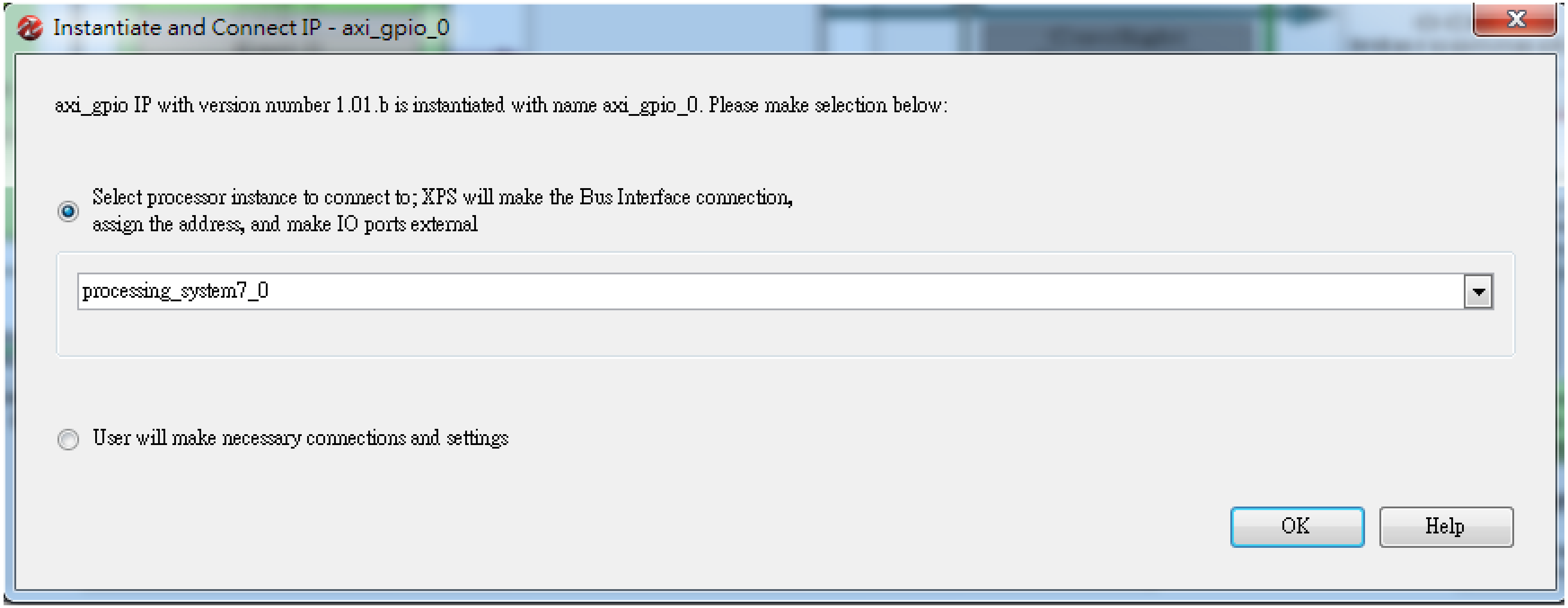


Click *Yes* when asked.

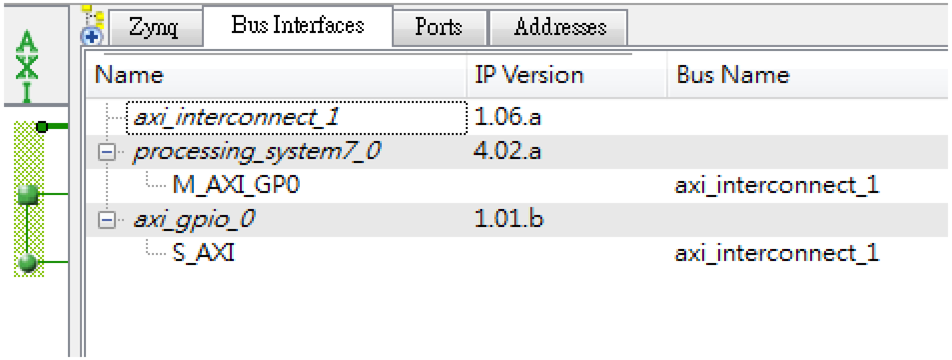


A configuration box will show up.

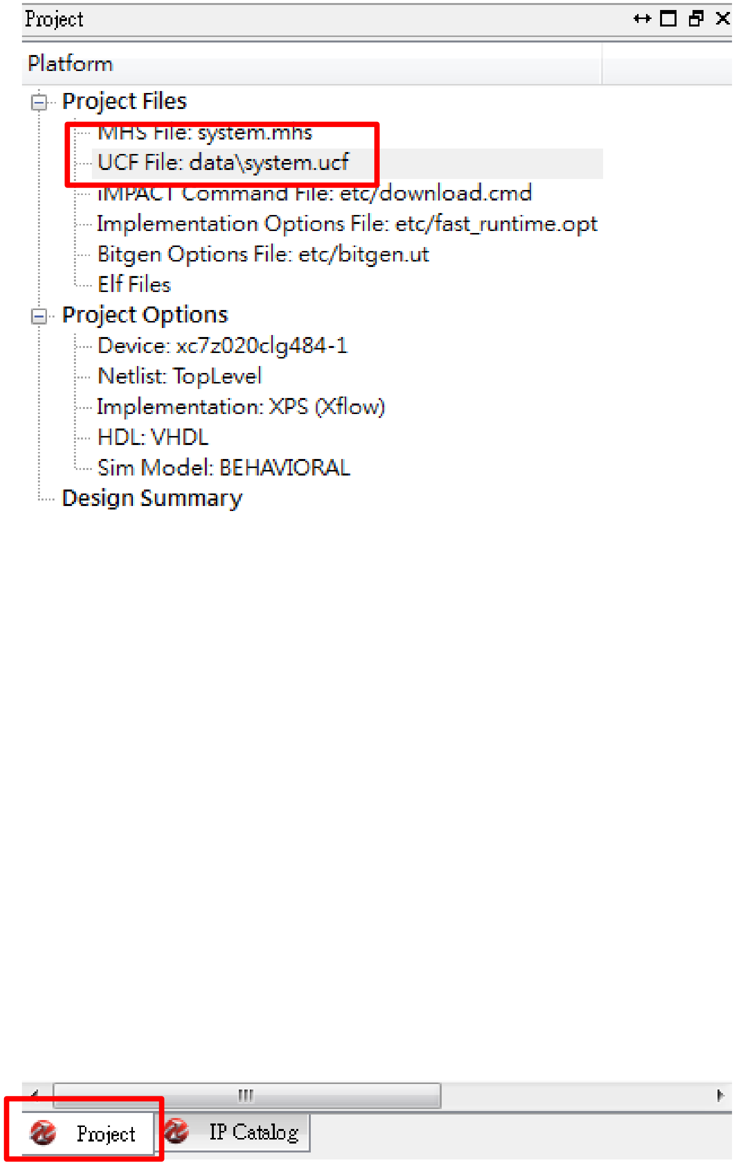
Modify Channel Width to 8. (Since we only have 8 LEDs)



Use the default settings.



Click on Bus Interfaces. You’ll find the connection is automatically established.

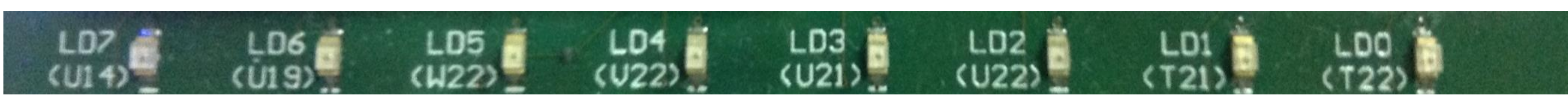


Click on *Project* tab

And double click the *UCF File*

system.ucf

```
NET axi_gpio_0_GPIO_IO_pin<0> LOC = T22  | IOSTANDARD=LVCMOS33;  # "LD0“
NET axi_gpio_0_GPIO_IO_pin<1> LOC = T21  | IOSTANDARD=LVCMOS33;  # "LD1“
NET axi_gpio_0_GPIO_IO_pin<2> LOC = U22  | IOSTANDARD=LVCMOS33;  # "LD2“
NET axi_gpio_0_GPIO_IO_pin<3> LOC = U21  | IOSTANDARD=LVCMOS33;  # "LD3“
NET axi_gpio_0_GPIO_IO_pin<4> LOC = V22  | IOSTANDARD=LVCMOS33;  # "LD4“
NET axi_gpio_0_GPIO_IO_pin<5> LOC = W22  | IOSTANDARD=LVCMOS33;  # "LD5“
NET axi_gpio_0_GPIO_IO_pin<6> LOC = U19  | IOSTANDARD=LVCMOS33;  # "LD6“
NET axi_gpio_0_GPIO_IO_pin<7> LOC = U14  | IOSTANDARD=LVCMOS33;  # "LD7"
```



Next...

- Export to SDK and generate BOOT.BIN, devicetree.dtb
  - Follow steps in the previous slides.
- Driver
  - Under OS, we need driver to connect user application and hardware.
- User application
  - An user end program

Hello World Driver

- Before we work on the driver for LED, let's take a look at a simple driver. This will help you understand:
  - How to compile a driver
  - How to insert (insmod) and remove(rmmod) a driver
  - How to print information in kernel.
- You need to compile kernel first since it need some information from the kernel. Also, the driver you use need to be compatible with your kernel. Or you'll get

```
zynq> insmod gpio.ko
[  110.420000] gpio: disagrees about version of symbol module_layout
insmod: can't insert 'gpio.ko': invalid module format
```

mkdir ~/win/driver/helloworld

cd ~/win/driver/helloworld

- We will use Makefile to compile driver

vi Makefile

```
KERN_SRC=/home/hadoop/linux-digilent
obj-m+=helloworld.o

all:
	make –C $(KERN_SRC) ARCH=arm M=`pwd` modules
clean:
	make –C $(KERN_SRC) ARCH=arm M=`pwd` clean
```

Note: the red region should be one tab, or an error will occur when compiling.

```
hadoop@ubuntu:~/win/driver/helloworld$ make
make: Nothing to be done for `all'.
```

helloworld.c

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/version.h>
```

Include files from kernel. Recall that we have define *KERN_SRC* in Makefile. The header file should exist under $(KERN_SRC)/include

```c
static int __init hello_init(void){
    printk(KERN_INFO "Hello World\n");
    return 0;
}
```

Operations related to insmod.

```c
static void __exit hello_exit(void){
    printk(KERN_INFO "Good Bye\n");
}
```

Operations related to rmmod.

```c
module_init(hello_init);
module_exit(hello_exit);
```

Define the functions will be called when insmod and rmmod.

```c
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hellow world driver");
MODULE_AUTHOR("Chunchen Tu.");
MODULE_VERSION("1.00a");
```

Driver information.

insmod, lsmod and rmmod

- Put the helloworld.ko into SD card. (Or use NFS)

  insmod helloworld.ko
- List the current driver

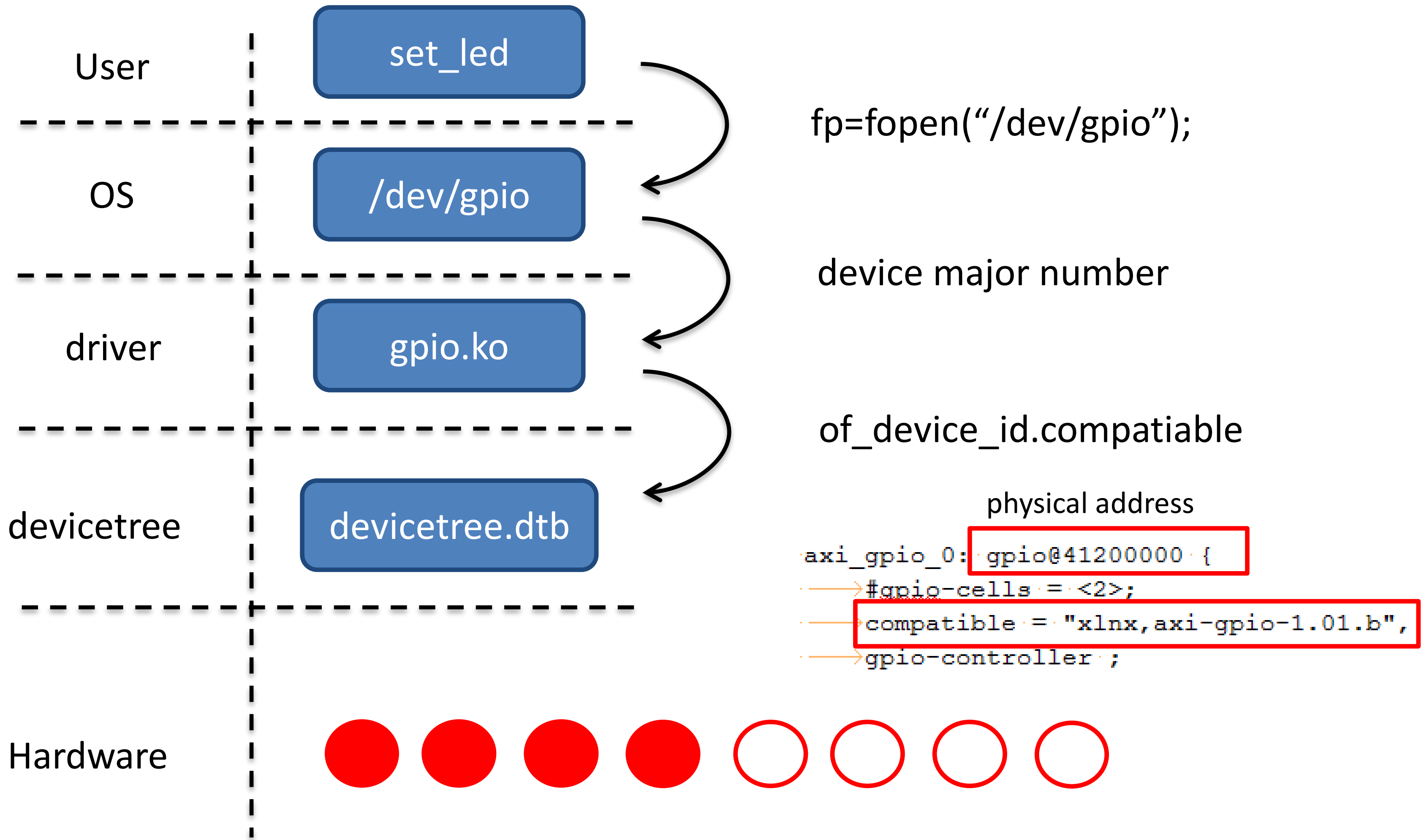
  lsmod
- Remove driver

  rmmod helloworld (Note: There is no “.ko” in the command)

```
zynq> insmod helloworld.ko
[ 1945.050000] Hello World
zynq> lsmod
helloworld 678 0 - Live 0xbf031000 (O)
zynq> rmmod helloworld
[ 1952.780000] Good Bye
```

But driver for LED is not that easy…

LED driver – From User to Hardware



File operations

```
/* File operations */
int gpio_open(struct inode *inode, struct file *filp)
{

int gpio_release(struct inode *inode, struct file *filp)
{

ssize_t gpio_read(struct file *filp, char __user *buf, size_t count,
        loff_t *f_pos)
{

ssize_t gpio_write(struct file *filp, const char __user *buf, size_t count,
        loff_t *f_pos)
{

struct file_operations gpio_fops = {
        .owner = THIS_MODULE,
        .open = gpio_open,
        .release = gpio_release
        .read = gpio_read,
        .write = gpio_write,
};
```

This is related to operations like (in C) fopen, fprintf, fscanf

For example (in C):

fp=fopen(/dev/gpio);

fprintf(fp,0xf0);

will link to gpio_write function in the kernel.

Driver statistics operations

```c
/* Driver /proc filesystem operations so that we can show some statistics */
static void *gpio_proc_seq_start(struct seq_file *s, loff_t *pos)
{

static void *gpio_proc_seq_next(struct seq_file *s, void *v, loff_t *pos)
{

static void gpio_proc_seq_stop(struct seq_file *s, void *v)
{

static int gpio_proc_seq_show(struct seq_file *s, void *v)
{

/* SEQ operations for /proc */
static struct seq_operations gpio_proc_seq_ops = {
        .start = gpio_proc_seq_start,
        .next = gpio_proc_seq_next,
        .stop = gpio_proc_seq_stop,
        .show = gpio_proc_seq_show
};
static int gpio_proc_open(struct inode *inode, struct file *file)
{

static struct file_operations gpio_proc_ops = {
        .owner = THIS_MODULE,
        .open = gpio_proc_open,
        .read = seq_read,
        .llseek = seq_lseek,
        .release = seq_release
};
```

Driver statistics. This will show up when you type: cat /proc/driver/gpio

Initialization and compatible

```c
#ifdef CONFIG_OF
static struct of_device_id gpio_of_match[] __devinitdata = {
        { .compatible = "xlnx,axi-gpio-1.01.b", },
        { /* end of table */}
};
MODULE_DEVICE_TABLE(of, gpio_of_match);
#else
#define gpio_of_match NULL
#endif /* CONFIG_OF */

static int gpio_remove(struct platform_device *pdev)
{

static int gpio_probe(struct platform_device *pdev)
{

static struct platform_driver gpio_driver = {
        .driver = {
                .name = MODULE_NAME,
                .owner = THIS_MODULE,
                .of_match_table = gpio_of_match,
        },
        .probe = gpio_probe,
        .remove = gpio_remove,
};

static void __exit gpio_exit(void)
{

static int __init gpio_init(void)
{

module_init(gpio_init);
module_exit(gpio_exit);
```

Driver initialization.
Note that the compatible should be consistent with the one in the device tree.

```
axi_gpio_0: gpio@41200000 {
        #gpio-cells = <2>;
        compatible = "xlnx,axi-gpio-1.01.b",
        gpio-controller ;
```

Flows of operations

insmod

gpio_init()

↓

gpio_probe(){
- driver initialization
- memory mapping

}

rmmod

gpio_exit()

↓

gpio_remove(){
- free resource
- detach memory mapping

}

Flows of operations

fprintf(fp,0xf0)

```
gpio_open ()
```

```
gpio_write(){
   copy_from_user(ker_buf,usr_data)
   iowrite8(ker_buf,virtual_addr)
}
```

```
gpio_release()
```

fread(fp,buf)

```
gpio_open ()
```

```
gpio_read(){
   ker_buf=ioread8(virtual_addr)
   copy_to_user(usr_buf,ker_buf)
}
```

```
gpio_release()
```

Address mapping

- We know that gpio is related to address 0x41200000. But we cannot access it directly.
  - For security reason.

```
axi_gpio_0: gpio@41200000 {
	#gpio-cells = <2>;
	compatible = "xlnx,axi-gpio-1.01.b",
	gpio-controller ;
```

- Map the physical address to virtual address.

```
gpio_dev->dev_physaddr = gpio_resource->start;
gpio_dev->dev_addrsize = gpio_resource->end -
        gpio_resource->start + 1;
if (!request_mem_region(gpio_dev->dev_physaddr,
        gpio_dev->dev_addrsize, MODULE_NAME)) {
        dev_err(&pdev->dev, "can't reserve i/o memory at 0x%08X\n",
                gpio_dev->dev_physaddr);
        status = -ENODEV;
        goto fail;
}
gpio_dev->dev_virtaddr = ioremap(gpio_dev->dev_physaddr,
        gpio_dev->dev_addrsize);
PDEBUG("gpio: mapped 0x%0x to 0x%0x\n", gpio_dev->dev_physaddr,
        (unsigned int)gpio_dev->dev_virtaddr);
```

In this case, we map 0x412000000 -> 0xe0880000
Once we want to operate gpio under kernel, we need to access it through virtual address.

```
[   36.680000] GPIO_INIT
[   36.690000] We have 1 resources
[   36.690000] devno is 0x3200000, pdev id is 0
[   36.690000] gpio: mapped 0x41200000 to 0xe0880000
[   36.700000] gpio 41200000.gpio: added GPIO driver successfully
```

gpio_write()

```c
ssize_t gpio_write(struct file *filp, const char __user *buf, size_t count,
        loff_t *f_pos)
{
        struct gpio_dev *dev = filp->private_data;

        int retval = 0;
        PDEBUG("GPIO_WRITE\n");
        transfer_size = count;
        PDEBUG("USER write 0x%02x",*(buf));
        iowrite8(0x00,dev->dev_virtaddr+0x4);
        iowrite8(*(buf),dev->dev_virtaddr);

        return count;
}
```

As the gpio_write is invoked (ex by fwrite), the OS will pass data and data size to gpio_write.
buf: buffer of data in from user.
count: data size

gpio_write()

```
ssize_t gpio_write(struct file *filp, const char __user *buf, size_t count,
        loff_t *f_pos)
{
        struct gpio_dev *dev = filp->private_data;

        int retval = 0;
        PDEBUG("GPIO_WRITE\n");
        transfer_size = count;
        PDEBUG("USER write 0x%02x",*(buf));
        iowrite8(0x00,dev->dev_virtaddr+0x4);
        iowrite8(*(buf),dev->dev_virtaddr);

        return count;
}
```

Also please check the usage of gpio IP. We should configure the tri-state register to 0 to make gpio operating in output mode.

Table 4: **Registers**

Base Address + Offset (hex)	Register Name	Access Type	Default Value (hex)	Description
C_BASEADDR + 0x00	GPIO_DATA	Read/Write	0x0	Channel 1 AXI GPIO Data Register
C_BASEADDR + 0x04	GPIO_TRI	Read/Write	0x0	Channel 1 AXI GPIO 3-state Register
C_BASEADDR + 0x08	GPIO2_DATA	Read/Write	0x0	Channel 2 AXI GPIO Data Register
C_BASEADDR + 0x0C	GPIO2_TRI	Read/Write	0x0	Channel 2 AXI GPIO 3-state Register

Table 7: **AXI GPIO Three-State Register Description**

Bits	Name	Core Access	Reset Value	Description
C_GPIOx_WIDTH - [1:0]	GPIOx_TRI	Read/Write	C_TRI_DEFAULT C_TRI_DEFAULT_2	AXI GPIO 3-state Control. Each I/O pin of the AXI GPIO is individually programmable as an input or output. For each of the bits: • 0 = I/O pin configured as output. • 1 = I/O pin configured as input.

mknod

- Next, we should create a device node

  mknod /dev/gpio c 50 0

  c: character device.

  50: major number

  0: minor number

```c
#define GPIO_MINOR        0

int gpio_major = 50;

gpio_dev->devno = MKDEV(gpio_major, GPIO_MINOR);
PDEBUG("devno is 0x%0x, pdev id is %d\n", gpio_dev->devno, GPIO_MINOR);
```

```
zynq> ls -l /dev/gpio
crw-r--r--    1 root     0          50,   0 Jan  1 00:33 /dev/gpio
```

major number

minor number

User application

vi set_led.c

```c
#include<stdio.h>

int main()
{
   FILE *fp=fopen("/dev/gpio","w");

   fprintf(fp,"%c",0xf0);
   fclose(fp);
   return 0;
}
```

It’s an easy example. Since there are only 8 leds, we only write a char into our device. For a larger memory example, please refer to the next slide.

Cross compile and test

Use the cross compiler (the one we use to compile kernel)

arm-xilinx-linux-gnueabi-gcc set_led.c –o set_led.out

you’ll get an binary file set_led
insert the driver file gpio.ko and execute set_led

```
zynq> insmod gpio.ko
[ 3186.710000] GPIO_INIT
[ 3186.720000] We have 1 resources
[ 3186.720000] devno is 0x3200000, pdev id is 0
[ 3186.720000] gpio: mapped 0x41200000 to 0xe0920000
[ 3186.730000] gpio 41200000.gpio: added GPIO driver successfully
zynq> ./set_led
[ 3190.480000] GPIO_OPEN
[ 3190.490000] GPIO_WRITE
[ 3190.490000] USER write f0
[ 3190.490000] GPIO_RELEASE
```

gpio_read()

- It’s easy just like gpio_write():
  - set the tri-state register.
  - ioread
  - print out
- Try it yourself