Beaglebone Black GPIO device driver

23ars

30 Apr 2016 GPL3

Rate this: 5.00 (3 votes)

This will be a simple article about Linux Device Drivers, in fact about a char driver that controls the GPIO (General Purpose Input-Output) port of the Beaglebone Black.

Introduction

This will be a simple article about Linux Device Drivers, in fact about a char driver that controls the GPIO (General Purpose Input-Output) port of the Beaglebone Black.

Background

Some time ago I worked on a project on Beaglebone Black that had to control some IOs, UART ports, also had to take some decisions and communicate with a PC via sockets. The interesting part at that project was the time constraint, the software had to take a decision or wait for an input at exactly the specified time. For example, it had to change the state of an output port from 0 to 1 with a frequency of 1 KHz, it had to change the value each millisecond. I managed to fulfill the time constraint by using code optimizations techniques and by writing some device drivers.

Using the code

The driver use IOCTLs for performing basic settings of the port, such as exporting a pin, set the direction input or output, use capture events such rising edge, falling edge etc. In the next paragraphs I will present the source code of some functions and also an example of how to use this device driver.

Data Types and IOCTL Options

For transferring data between user space and kernel space and also for IOCTL I used a structure that contains two buffers for read and write operations, the number of the pin that is controlled and the interrupt number, if interrupts are used for reading data.

Hide Copy Code

```
struct bbbgpio_ioctl_struct
{
    u16 gpio_number;
    u8 write_buffer;
    u8 read_buffer;
    int irq_number;
};
```

The settings that should be made before reading and writing data, driver's work mode etc. are implemented using IOCTLs. All the requests that are defined for this device driver are:

Hide Copy Code

```
#define IOCBBBGPIORP       _IOW(_IOCTL_MAGIC,1,struct bbbgpio_ioctl*)     /*IOCTL used for
registering PIN*/
#define IOCBBBGPIOUP       _IOW(_IOCTL_MAGIC,2,struct bbbgpio_ioctl*)     /*IOCTL used for
unregistering PIN*/
#define IOCBBBGPIOWR       _IOW(_IOCTL_MAGIC,3,struct bbbgpio_ioctl*)      /*write data to
register*/
#define IOCBBBGPIORD       _IOR(_IOCTL_MAGIC,4,struct bbbgpio_ioctl*)      /*read from
register*/
#define IOCBBBGPIOSD       _IOW(_IOCTL_MAGIC,5,struct bbbgpio_ioctl*)      /*set
direction*/
#define IOCBBBGPIOSL0      _IOW(_IOCTL_MAGIC,6,struct bbbgpio_ioctl*)      /*set low
detect*/
#define IOCBBBGPIOSH1      _IOW(_IOCTL_MAGIC,7,struct bbbgpio_ioctl*)      /*set high
detect*/
#define IOCBBBGPIOSRE      _IOW(_IOCTL_MAGIC,8,struct bbbgpio_ioctl*)      /*set rising
edge*/
#define IOCBBBGPIOSFE      _IOW(_IOCTL_MAGIC,9,struct bbbgpio_ioctl*)      /*set falling
edge*/
#define IOCBBBGPIOSIN      _IOW(_IOCTL_MAGIC,10,struct bbbgpio_ioctl*)      /*enable gpio
interrupt*/
#define IOCBBBGPIOSBW      _IOW(_IOCTL_MAGIC,11,struct bbbgpio_ioctl*)      /*enable gpio
busy wait mode*/
```

Writing Data

The easiest part to implement at this device driver was the function that set the output state of the port, the function should take a value from user space and write it in the port's register.

Hide Copy Code

```
static ssize_t
bbbgpio_write(struct file *filp, const char __user *buffer, size_t length, loff_t *offset)
{
    if (mutex_trylock(&bbbgpiodev_Ptr->io_mutex) == 0) {
        driver_err("%s:Mutex not free!\n",DEVICE_NAME);
        return -EBUSY;
    }

    if (copy_from_user(&ioctl_buffer,buffer,sizeof(struct bbbgpio_ioctl_struct)) != 0) {
        driver_err("%s:Could not copy data from userspace!\n",DEVICE_NAME);
        mutex_unlock(&bbbgpiodev_Ptr->io_mutex);
        return -EINVAL;
    }
    gpio_set_value(ioctl_buffer.gpio_number,ioctl_buffer.write_buffer);
    mutex_unlock(&bbbgpiodev_Ptr->io_mutex);
    return 0;
}
```

This function will try to lock a mutex, to prevent concurrent access to shared data and if it will fail, will return **EBUSY.** If the value returned by **mutex_trylock** was 0, SUCCESS, buffer's content will be copied from userspace and if the operation succeeds, the output value will be updated with the value from write_buffer.

Reading Data

A little more challenging part was to implement the read functionality because the device driver had to work in busy-wait mode or using interrupts.

Busy-wait Mode

In busy-wait, the driver will read the value of the port and it will send it to user space. This mode was designed to be used when data should be captured continuously, without using any triggers.

Hide Copy Code

```
static ssize_t
bbbgpio_read(struct file *filp,char __user *buffer,size_t length,loff_t *offset)
{

    if (mutex_trylock(&bbbgpiodev_Ptr->io_mutex) == 0) {
        driver_err("%s:Mutex not free!\n",DEVICE_NAME);
        return -EBUSY;
    }
    if (copy_from_user(&ioctl_buffer,buffer,sizeof(struct bbbgpio_ioctl_struct)) != 0) {
        driver_err("%s:Could not copy data from userspace!\n",DEVICE_NAME);
        mutex_unlock(&bbbgpiodev_Ptr->io_mutex);
        return -EINVAL;
    }

    ioctl_buffer.read_buffer=gpio_get_value(ioctl_buffer.gpio_number);
    if (copy_to_user(buffer,&ioctl_buffer,sizeof(struct bbbgpio_ioctl_struct)) !=0 ) {
        driver_err("\t%s:Cout not write values to user!\n",DEVICE_NAME);
        mutex_unlock(&bbbgpiodev_Ptr->io_mutex);
        return -EINVAL;
    }
    mutex_unlock(&bbbgpiodev_Ptr->io_mutex);
    return 0;

}
```

The read function is somehow similar with the write. Firstly, will try to lock a mutex, and if the operation succeeds, it will copy from userspace the buffer's value. This operation is needed because I wanted to know what pin I will read. Next, it will read the pin's value and after that will update the buffer and copy it back to userspace.

Interrupt Mode

Interrupt mode was designed because I wanted the driver to react on events and start to capture data only when a transition from HIGH to LOW or rising/falling edge is detected. Data is captured in the ISR (interrupt service routine), is saved in a ring buffer and only when **ioctl** with **IOCBBBGPIORD** request is called, it will be transferred to userspace.

Hide Copy Code

```
static irq_handler_t
irq_handler(int irq,void *dev_id,struct pt_regs *regs)
{
    u16 io_number=(u16)dev_id;
    struct bbb_data_content content;
    if (mutex_trylock(&bbbgpiodev_Ptr->io_mutex) == 0) {
        driver_err("%s:Mutex not free!\n",DEVICE_NAME);
        goto exit_interrupt;
    }

    content.data=gpio_get_value(io_number);
    content.dev_id=io_number;
    bbb_buffer_push(&bbb_data_buffer,content);

    mutex_unlock(&bbbgpiodev_Ptr->io_mutex);
    driver_info("Interrup handler executed!\n");
exit_interrupt:{

        return (irq_handler_t) IRQ_HANDLED;
    }
}
```

When the interrupt service routine is called, the mutex will be locked, data will be read from the pin and saved in the ring buffer **bbb_data_buffer**.

IOCBBBGPIORD ioctl request is handled in the function **static long bbbgpio_ioctl(struct file *file, unsigned int ioctl_num ,unsigned long ioctl_param)** where is checked if the ring buffer is not empty and, if is not, then data will be take from it and copied to user space.

Example Application

An example of how to use this device driver can be found HERE.

In the Makefile of the project I also added some steps for installing & uninstalling the driver.

Hide Copy Code

```
FILE=bbbgpio
obj-m += $(FILE).o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

install:
    sudo insmod ./$(FILE).ko
    cp bbbgpio_ioctl.h /usr/include/
run:
    dmesg
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
    sudo rmmod $(FILE)
```

Points of Interest

This project has bought me a lot of fun, one or two sleepless nights and a little bit of frustration because I didn't want to use **linux/gpio.h** but to write values directly to registers. I implemented the read & write by directly accessing registers but I was stucked at detecting the interrupt number, I couldn't probe and register the interrupt for edge detecting.

Full source code can be found HERE.

History

- Initial commit
- Working version

License

Share

About the Author

23ars
Engineer
Romania

I'm a software engineer and I have 3 years experience in software development. I worked for one year as Java Developer and now I'm working as embedded C software developer in automotive domain. I like a lot embedded software development and I'm passionate about Linux.

Comments and Discussions

You must Sign In to use this message board.

Search Comments

-- There are no messages in this forum --