September 10, 2016 maquefel Articles

Using gpio-generic and irq_chip_generic subsystems for gpio driver

Intoduction

This article is a logical continuation of the previous one and it is recommended to stady after reading the previous material. The current note is necessary for understanding the subsequent material, a further understanding gpio subsystem as a whole and helps with the development of own gpio drivers. The provisions of this article have not only apply to our virtual driver gpio, but also to any mmio gpio driver in general. It will focus on "optimizing" referred to in the preceding article.

Purpose

Before proceeding to a more interesting and useful things, it is necessary to do some code cleanup. As mentioned earlier the amount of code can be reduced. For this we will use two subsystems (drivers) gpio-generic (gpio-mmio since 4.7) and irq_chip_generic.

According to the principles of the linux kernel it is better to sacrifice performance for the sake of clarity and lack of recurring code. And the use of existing implementations in linux kernel, certainly serves this purpose.

Implementation

pci_ids

> Small offtopic.

Initially, during the transition to bgpio sybsytem, a small successful experiment of supporting 8, 16 and 32 inputs in virtual device driver was conducted. The final code can be found [here] (https://github.com/maquefel/virtual_gpio_basic/tree/pci_ids). This result was achieved thanks to a small, only 7 lines, ivshmem modification to add the parameters sub-vendor-id and a sub-device-id passed to the pci device initialization (included in the patch branch for version qemu 2.5.1.1).

Using gpio-mmio subsystem

> This paramer depends on **CONFIG_GPIO_GENERIC** kernel config option.

Let's start with a simple things - driver that was created for different MMIO gpio devies, and later part of which was actively used by some drivers. Subsystem was represented by Anton Vorontsov in 2010 (https://lkml.org/lkml/2010/8/25/303). In his patch, he announced the following features for the driver:

- Support of 8/16/32/64 bit registers
- Support for controllers gpio with set/clear registers
- Support for controllers gpio with data only register
- Support for big-endian

In genral, the driver saves us from implementing such functions such as the state of the pins and the choice of the pins direction.

To use it - it is just enough to pass the size of bank and the data register, the rest is optional and depends on the specific implementation of gpio controller. Initialization function takes as parameters:

- Status register
- Set status register
- Clean status register
- Switching contact to the output state register
- Switching contact to the input state register

So the following situations are possible:

- Read-only "data" register, "set" register to set the state and "clr" register to clear the state
- Read-only "data" register, "set" register to set and clear the state
- "Data" register for all above

The same is true for setting directions. Either register to set direction output or as input. Or the lack of the possibility to switch the contact into a state of output or input.

It is possible to simulate any of the above behavior. For our purposes it is enough to conform the number 3 and the setting the state of pin as the output, with default state of the contact as input.

bgpio_init

```
#if IS_ENABLED(CONFIG_GPIO_GENERIC)

int bgpio_init(struct gpio_chip *gc,
               struct device *dev,
               unsigned long sz,
               void __iomem *dat,
               void __iomem *set,
               void __iomem *clr,
               void __iomem *dirout,
               void __iomem *dirin,
```

It is worth noting that as the size bgpio_init not accept the number of pins, but multiple parameter 8, ie ngpio/8.

```
    err = bgpio_init(&vg->chip, dev, BITS_TO_BYTES(VIRTUAL_GPIO_NR_GPIOS),
                     data, NULL, NULL, dir, NULL, 0);
```

Usage of irq_chip_generic subsystem

> This driver depends on ** GENERIC_IRQ_CHIP ** kernel config option, which is a disadvantage, because it is not possible to enable this option through menuconfig or oldconfig.

Now let's look at a little more difficult part. irq_chip_generic was introduced in kernel version [v3.0-rc1] (https://github.com/torvalds/linux/commit/7d8280624797bbe2f5170bd3c85c75a8c9c74242) and performs functions similar to gpio-mmio, ie provides a standard implementation for many occasions of irq_chip's.

Standard read/write register function are 32 bit, so it was one of the reasons I decided to abandon the 8/16 bit versions of driver, yet [we have the possibility to provide our own functions to read/write or define standard] (https://lkml.org/lkml/2014/11/7/68) (eg ioread8, iowrite8).

A try on using irq_chip_genric togather with gpiochip_irqchip_add and gpiochip_set_chained_irqchip

Memory allocation for irq_chip_generic and initialization is done by irq_alloc_generic_chip. Function, in addition to the trivial parameters, also requires irq_base parameter, which, generally speaking, we do not know untill gpiochip_irqchip_add call, which in turn requires initialized struct irq_chip. But irq_alloc_generic_chip is responsible only for the allocation and initialization of certain parameters, so that we can pass 0 as irq_base parameter and assign the actual value after the gpiochip_irqchip_add function call.

```
    gc = irq_alloc_generic_chip(VIRTUAL_GPIO_DEV_NAME, 1, 0, vg->data_base_addr,
```

For usage it is sufficient to assign pointers to standard mask/unmask function, confirmation of interruption function and registers. Interrupt type setting function have remained unchanged, as well as our interrupt handler.

```
    ct->chip.irq_ack = irq_gc_ack_set_bit;
    ct->chip.irq_mask = irq_gc_mask_clr_bit;
    ct->chip.irq_unmask = irq_gc_mask_set_bit;
    ct->chip.irq_set_type = virtual_gpio_irq_type;
```

Registers for irq enable and ack;

```
    ct->regs.ack = VIRTUAL_GPIO_INT_EOI;
    ct->regs.mask = VIRTUAL_GPIO_INT_EN;
```

We still handle the interrupt type registers by ourselves, in virtual_gpio_irq_type function.

Accordingly, we pass irq_chip instance wich was allocated in irq_chip_generic->chip_types[0] during the initialization to the gpiochip_irqchip_add gpiochip_set_chained_irqchip functions (irq_chip_generic can have several types of irq_chip associated with the same subset irq).

Then we use the resulting irq_base and finish irq_chip_generic settings.

```
irq_setup_generic_chip(gc, 0, 0, 0, 0);

gc->irq_cnt = VIRTUAL_GPIO_NR_GPIOS;
gc->irq_base = vg->chip.irq_base;

u32 msk = IRQ_MSK(32);
u32 i;

for (i = gc->irq_base; msk; msk >>= 1, i++) {
    struct irq_data *d = irq_get_irq_data(i);
    d->mask = 1 << (i - gc->irq_base);
```

We deliberately pass 0 as the msk parameter, to avoid irq numbers re-initialization.

irq_chip_generic used for quite a serious irq controllers in most platforms, so irq mask registers offsets are calculated in advance not to waste time on the calculation of the masks in the process. As we passed 0 as a parameter in irq_setup_generic_chip function, initializing the masks by ourselfs.

There is a remaining one problem with irq_set_chip_data (which is nothing else but `irq_data.chip_data = (void *) struct irq_chip_generic;`). The fact is that gpiolib also relies on `void *chip_data` and believes that there should always be a pointer to a struct gpio_chip there (http://lxr.free-electrons.com/source/drivers/gpio/gpiolib.c#L1138). The problem solved by providing our own implementations of irq_request_resources and irq_release_resources functions, and instead of the standard gpiochip_irq_reqres and gpiochip_irq_relres.

The source code for this example.

An alternative approach

Strictly speaking the above manipulations are not clear, and there may be some difficulty with their understanding. So let's try to get rid of gpiochip_irqchip_add.

First and foremost, we ask for irq_base ourself:

```
irq_base = irq_alloc_descs(-1, 0, vg->chip.ngpio, 0);
```

As we remember from the previous article, our case is a linear (function irq_domain_add_simple does almost the same thing, but it is marked as legacy).

```
vg->irq_domain = irq_domain_add_linear(0, vg->chip.ngpio, &irq_domain_simple
irq_domain_associate_many(vg->irq_domain, irq_base, 0, vg->chip.ngpio);
```

The next step has remained virtually unchanged and it is binding gpio_chip to irq_chip:

```
vg->chip.irqdomain = vg->irq_domain;
gpiochip_set_chained_irqchip(&vg->chip, &ct->chip, pdev->irq, NULL);
```

The only moment - is that for file "edge" was available in the according sysfs directory of gpio([gpiolib-sysfs] (http://lxr.free-electrons.com/source/drivers/gpio/gpiolib-sysfs.c#L353)), we must specify the translation function to_irq:

```
vg->chip.to_irq = virtual_gpio_to_irq;
```

Conclusion

Through the use of existing subsystems code slightly decreased. In the understanding of the driver as well it has become easier, because the use of these subsystems is quite common, and we see immediately that this is a standard driver without pitfalls.

Suppositive downside should be considered the depency on the ** CONFIG_GPIO_GENERIC ** and ** GENERIC_IRQ_CHIP ** kernel options, but the first option is easily enabled in the configuration, and second, most likely, already have been enabled.

Materials recommended for further reading:
1. High-level IRQ flow handlers
2. Linux Kernel IRQ Domain
3. Edge Triggered Vs Level Triggered interrupts

The source code, Makefile and the README:
https://github.com/maquefel/virtual_gpio_basic/tree/v4.6

gpio, irq, kernel module, linux, linux kernel

QEMU ivshmem Introducing interrupt capable virtual gpio concept

Don’t fear the bricking (unbricking Unmatched via JTAG)