CREACION PLATFORM DRIVER GPIO SWITCH

June 2, 2017

Part I
Source Code

```
/*
 * Based on Xilinx GPIO driver
 * Copyright 2008 Xilinx, Inc.
 *
 * This program is free software; you can redistribute
     it and/or modify
 * it under the terms of the GNU General Public
    License version 2
 * as published by the Free Software Foundation.
 *
 * You should have received a copy of the GNU General
    Public License
 * along with this program; if not, write to the Free
    Software
 * Foundation, Inc., 59 Temple Place, Suite 330,
    Boston, MA  02111-1307  USA
 */
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/of_device.h>
#include <linux/of_platform.h>
#include <linux/of_gpio.h>
#include <linux/io.h>
#include <linux/gpio.h>
#include <linux/module.h>
#include <linux/slab.h>

struct gcq_gpio_switch_chip {
        struct of_mm_gpio_chip mmchip;
        spinlock_t gpio_lock;   /* Lock used for
           synchronization */
};

static int gcq_gpio_switch_get(struct gpio_chip *gc,
   unsigned int gpio) {
        int bank;
        int g;
        struct of_mm_gpio_chip *mm_gc =
           to_of_mm_gpio_chip(gc);

        bank = gpio >> 3;
```

```
        g = gpio%8;
    printk(KERN_INFO "gcq_gpio_get\n");
        return (ioread8(mm_gc->regs + bank) >> g) & 1;
}

/*
        status = chip->get_direction(chip, offset);
        if (status > 0) {
                //GPIOF_DIR_IN, or other positive
                status = 1;
                clear_bit(FLAG_IS_OUT, &desc->flags);
        }
        if (status == 0) {
                // GPIOF_DIR_OUT
                set_bit(FLAG_IS_OUT, &desc->flags);
        }
*/
static int gcq_gpio_switch_get_direction(struct
   gpio_chip *gc, unsigned int gpio) {
        printk(KERN_INFO "gcq_gpio_get_dir\n");
        return 1;
}

static int gcq_gpio_switch_probe(struct
   platform_device *pdev)  {
        //nodo del dts que se conecta al driver
         struct device_node *node = pdev->dev.of_node;
        int status;
     const u32 *tree_info;
        struct gcq_gpio_switch_chip *gcq_gc;

        gcq_gc = devm_kzalloc(&pdev->dev, sizeof(*
           gcq_gc), GFP_KERNEL);
        if (!gcq_gc)
                return -ENOMEM;

         //inicializamos el lock
        spin_lock_init(&gcq_gc->gpio_lock);
         //valor por defecto de 8 GPIO
     gcq_gc->mmchip.gc.ngpio = 8;
         //leemos el campo del dts xlnx,gpio-width
         tree_info = of_get_property(node,"xlnx,gpio-
            width",NULL);
        if (tree_info)
                gcq_gc->mmchip.gc.ngpio = *tree_info;
```

```
        //creamos puntero a nuestras funciones, mas
            adelante veremos como las llama el kernel
        gcq_gc->mmchip.gc.get_direction =
            gcq_gpio_switch_get_direction;
        gcq_gc->mmchip.gc.get = gcq_gpio_switch_get;

        //registramos el dispositivo GPIO
        /* Call the OF gpio helper to setup and
            register the GPIO device */
        status = of_mm_gpiochip_add(node, &gcq_gc->
            mmchip);
        if (status) {
                dev_err(&pdev->dev, "Failed adding
                    memory mapped gpiochip\n");
                return status;
        }
        platform_set_drvdata(pdev, gcq_gc);
    return status;
}

static int gcq_gpio_switch_remove(struct
    platform_device *pdev) {
        struct gcq_gpio_switch_chip *gcq_gc =
            platform_get_drvdata(pdev);
        of_mm_gpiochip_remove(&gcq_gc->mmchip);
        return 0;
}

static struct of_device_id gcq_gpio_switch_of_match[]
    = {
        { .compatible = "dte,gcq-gpio-switch" },
        { /* end of list */ },
};

MODULE_DEVICE_TABLE(of, gcq_gpio_switch_of_match);

static struct platform_driver gcq_gpio_switch_driver =
    {
        .driver = {
                .name   = "gcq-gpio-switch",
                .of_match_table = of_match_ptr(
                    gcq_gpio_switch_of_match),
        },
        .probe          = gcq_gpio_switch_probe,
        .remove         = gcq_gpio_switch_remove,
};
```

```
static int __init gcq_gpio_switch_init(void) {
        return platform_driver_register(&
           gcq_gpio_switch_driver);
}
subsys_initcall(gcq_gpio_switch_init);

static void __exit gcq_gpio_switch_exit(void) {
        platform_driver_unregister(&
           gcq_gpio_switch_driver);
}
module_exit(gcq_gpio_switch_exit);

MODULE_AUTHOR("GCQ");
MODULE_DESCRIPTION("Testing␣GPIO␣SWITCH␣driver");
MODULE_LICENSE("GPL");
```

Part II
DTS

```
switch0: gpio@B8000000 {
                compatible = "dte,gcq-gpio-switch";
                reg = <0xB8000000 0x1>;
                #gpio-cells = <2>;
                gpio-controller;
                xlnx,gpio-width = <8>;
        };
```

Part III
Registro Driver

```
/*
 * /drivers/base/platform_device.h
*/
#define platform_driver_register(drv)
    __platform_driver_register(drv, THIS_MODULE)

/*
 * /include/linux/export.h
*/
#ifdef MODULE
extern struct module __this_module;
#define THIS_MODULE (&__this_module)
#else
#define THIS_MODULE ((struct module *)0)
#endif

/*
 * /drivers/base/platform.c
*/
/**
 * __platform_driver_register - register a driver for
     platform-level devices
 * @drv: platform driver structure
 * @owner: owning module/driver
 */
int __platform_driver_register(struct platform_driver
    *drv, struct module *owner) {
         drv->driver.owner = owner;
         drv->driver.bus = &platform_bus_type;
         drv->driver.probe = platform_drv_probe;
         drv->driver.remove = platform_drv_remove;
         drv->driver.shutdown = platform_drv_shutdown;
         return driver_register(&drv->driver);
}

/*
 * /drivers/base/driver.c
*/
/*
.driver = {
                .name    = "gcq-gpio-switch",
```

```
                .of_match_table = of_match_ptr(
                    gcq_gpio_switch_of_match),
                ---------------------------------
                .bus = {
                                .name           = "
                                    platform",
                                .dev_groups     =
                                    platform_dev_groups
                                    ,
                                .match          =
                                    platform_match, //
                                    bind platform
                                    device to platform
                                    driver.
                                .uevent         =
                                    platform_uevent,
                                .pm             = &
                                    platform_dev_pm_ops
                                    ,
                                }
        }
*/
int driver_register(struct device_driver *drv) {
        [...]
        //se mira si existe ya un driver con el mismo
            nombre en el platform bus
        other = driver_find(drv->name, drv->bus);
        if (other) {
                printk(KERN_ERR "Error:␣Driver␣'%s'␣is
                    ␣already␣registered,␣"
                         "aborting...\n", drv->name);
                return -EBUSY;
        }
        /*
                principalmente en bus_add_driver se
                    llama a:
                1. module_add_driver(drv->owner, drv)
                2. driver_create_file(drv, &
                    driver_attr_uevent)
        */
        ret = bus_add_driver(drv);
        if (ret)
                return ret;

        [...]
```

```
        return ret;
}

void module_add_driver(struct module *mod, struct
    device_driver *drv){
        struct module_kobject *mk = NULL;
        [...]
        mk = &mod->mkobj;
        /**
                 *      sysfs_create_link - create
                    symlink between two objects.
                 *      @kobj:  object whose directory
                     we're creating the link in.
                *       @target:        object we're
                    pointing to.
                *       @name:          name of the
                    symlink.
        */
        no_warn = sysfs_create_link(&drv->p->kobj, &mk
            ->kobj, "module");
        /*
                * driver_name = "%s:%s",drv->bus->name
                    ,drv->name
        */
        driver_name = make_driver_name(drv);
        if (driver_name) {
                /*
                        * mk->drivers_dir =
                            kobject_create_and_add("
                            drivers", &mk->kobj);
                */
                module_create_drivers_dir(mk);
                no_warn = sysfs_create_link(mk->
                    drivers_dir, &drv->p->kobj,
                    driver_name);
                kfree(driver_name);
        }

}

/**
 * driver_create_file - create sysfs file for driver.
 * @drv: driver.
 * @attr: driver attribute descriptor.
 */
int driver_create_file(struct device_driver *drv,
```

```
                        const struct driver_attribute *
                           attr) {
        int error;
        if (drv)
                error = sysfs_create_file(&drv->p->
                   kobj, &attr->attr);
        else
                error = -EINVAL;
        return error;
}
EXPORT_SYMBOL_GPL(driver_create_file);
```

Part IV
deteccion de GPIO

```
/**
 * of_mm_gpiochip_add - Add memory mapped GPIO chip (
    bank)
 * @np:          device node of the GPIO chip
 * @mm_gc:       pointer to the of_mm_gpio_chip
    allocated structure
 *
 * To use this function you should allocate and fill
    mm_gc with:
 *
 * 1) In the gpio_chip structure:
 *    - all the callbacks
 *    - of_gpio_n_cells
 *    - of_xlate callback (optional)
 *
 * 3) In the of_mm_gpio_chip structure:
 *    - save_regs callback (optional)
 *
 * If succeeded, this function will map bank's memory
    and will
 * do all necessary work for you. Then you'll able to
    use .regs
 * to manage GPIOs from the callbacks.
 */
int of_mm_gpiochip_add(struct device_node *np,
                       struct of_mm_gpio_chip *mm_gc)
                          {
        int ret = -ENOMEM;
        struct gpio_chip *gc = &mm_gc->gc;
        /**
                 * kstrdup - allocate space for and
                    copy an existing string
                 * @s: the string to duplicate
                 * @gfp: the GFP mask used in the
                    kmalloc() call when allocating
                    memory
         */
        gc->label = kstrdup(np->full_name, GFP_KERNEL)
           ;
        if (!gc->label)
                goto err0;
```

```
        //mapeamos la mamoria ver mas adelante.
        mm_gc->regs = of_iomap(np, 0);
        if (!mm_gc->regs)
                goto err1;
        gc->base = -1;
        //se puede definir la funcion save_regs en el
            driver para dar valor inicial a los
            registros
        if (mm_gc->save_regs)
                mm_gc->save_regs(mm_gc);
        mm_gc->gc.of_node = np;
        ret = gpiochip_add(gc);
        if (ret)
                goto err2;
        return 0;
err2:
        iounmap(mm_gc->regs);
err1:
        kfree(gc->label);
err0:
        pr_err("%s: GPIO chip registration failed with
            status %d\n",
               np->full_name, ret);
        return ret;
}
EXPORT_SYMBOL(of_mm_gpiochip_add);
```

mapeado memoria

```
void __iomem *of_iomap(struct device_node *np, int
    index) {
        struct resource res;
        if (of_address_to_resource(np, index, &res))
                return NULL;
        //reserva desde (res.start) hasta (res->end -
            res->start + 1)
        return ioremap(res.start, resource_size(&res))
            ;
}
EXPORT_SYMBOL(of_iomap);

int of_address_to_resource(struct device_node *dev,
    int index,
                           struct resource *r) {
        const __be32    *addrp;
        u64             size;
        unsigned int    flags;
        const char      *name = NULL;
        addrp = of_get_address(dev, index, &size, &
            flags);
        if (addrp == NULL)
                return -EINVAL;
        /* Get optional "reg-names" property to add a
            name to a resource */
        of_property_read_string_index(dev, "reg-names"
            , index, &name);
        return __of_address_to_resource(dev, addrp,
            size, flags, name, r);
}
EXPORT_SYMBOL_GPL(of_address_to_resource);

const __be32 *of_get_address(struct device_node *dev,
    int index, u64 *size,
                    unsigned int *flags) {
        const __be32 *prop;
        unsigned int psize;
        struct device_node *parent;
        struct of_bus *bus;
        int onesize, i, na, ns;
        /* Get parent & match bus type */
        // devuelve el nodo padre en dts
        parent = of_get_parent(dev);
```

```
if (parent == NULL)
        return NULL;
/*
        *en este punto bus sera
        {
        .name = "default",
        .addresses = "reg",
        .match = NULL,
        .count_cells =
           of_bus_default_count_cells,
        .map = of_bus_default_map,
        .translate = of_bus_default_translate,
        .get_flags = of_bus_default_get_flags,
        }
*/
bus = of_match_bus(parent);
//na = #address-cells
//ns = #size-cells
bus->count_cells(dev, &na, &ns);
of_node_put(parent);
if (!OF_CHECK_ADDR_COUNT(na))
        return NULL;
/* Get "reg" or "assigned-addresses" property
   */
// prop = <0xB8000000 0x1>
prop = of_get_property(dev, bus->addresses, &
   psize);
if (prop == NULL)
        return NULL;
// psize son 2 elementos de u32, 8 bytes
// psize = 8/4 = 2
psize /= 4;
//onesize = 2
onesize = na + ns;
for (i = 0; psize >= onesize; psize -= onesize
   , prop += onesize, i++)
        //index en nuestro caso es 0
        if (i == index) {
                if (size)
                        //espacio de memoria
                           necesario le a
                           partir de prop+na,
                           donde comienza ns
                        *size = of_read_number
                           (prop + na, ns);
                if (flags)
```

```
                                        *flags = bus->
                                            get_flags(prop);
                                //devolvemos <0xB8000000 0x1>
                                return prop;
                        }
                return NULL;
}
EXPORT_SYMBOL(of_get_address);


static int __of_address_to_resource(struct device_node
    *dev,
                const __be32 *addrp, u64 size,
                    unsigned int flags,
                const char *name, struct resource *r)
                    {
        u64 taddr;
        if ((flags & (IORESOURCE_IO | IORESOURCE_MEM))
            == 0)
                return -EINVAL;
        //llama a __of_translate_address con rprop = "
            ranges"
        //en este caso taddr = 0xB8000000
        taddr = of_translate_address(dev, addrp);
        if (taddr == OF_BAD_ADDR)
                return -EINVAL;
        memset(r, 0, sizeof(struct resource));
        if (flags & IORESOURCE_IO) {
                unsigned long port;
                port = pci_address_to_pio(taddr);
                if (port == (unsigned long)-1)
                        return -EINVAL;
                r->start = port;
                r->end = port + size - 1;
        } else {
                r->start = taddr;
                r->end = taddr + size - 1;
        }
        r->flags = flags;
        r->name = name ? name : dev->full_name;
        return 0;
}

static u64 __of_translate_address(struct device_node *
    dev,
                                    const __be32 *
```

```
                                in_addr, const
                                char *rprop) {
struct device_node *parent = NULL;
struct of_bus *bus, *pbus;
__be32 addr[OF_MAX_ADDR_CELLS];
int na, ns, pna, pns;
u64 result = OF_BAD_ADDR;

pr_debug("OF: ** translation for device %s **\
    n", of_node_full_name(dev));
/* Increase refcount at current level */
of_node_get(dev);
/* Get parent & match bus type */
parent = of_get_parent(dev);
if (parent == NULL)
        goto bail;
bus = of_match_bus(parent);
/* Count address cells & copy address locally
    */
bus->count_cells(dev, &na, &ns);
if (!OF_CHECK_COUNTS(na, ns)) {
        pr_debug("OF: Bad cell count for %s\n"
            , of_node_full_name(dev));
        goto bail;
}
//copia en addr la direccion in_addr = <0
    xB8000000 0x1>
memcpy(addr, in_addr, na * 4);
pr_debug("OF: bus is %s (na=%d, ns=%d) on %s\n
    ",
     bus->name, na, ns, of_node_full_name(
        parent));
of_dump_addr("OF: translating address:", addr,
     na);
/* Translate */
for (;;) {
        /* Switch to parent bus */
        of_node_put(dev);
        dev = parent;
        parent = of_get_parent(dev);
        /* If root, we have finished */
        if (parent == NULL) {
                pr_debug("OF: reached root
                    node\n");
                //lee el valor desde addr na
                    bytes
```

```
                        result = of_read_number(addr,
                            na);
                        break;
                }
                /* Get new parent bus and counts */
                pbus = of_match_bus(parent);
                pbus->count_cells(dev, &pna, &pns);
                if (!OF_CHECK_COUNTS(pna, pns)) {
                        printk(KERN_ERR "prom_parse:␣
                            Bad␣cell␣count␣for␣%s\n",
                                of_node_full_name(dev))
                                    ;
                        break;
                }
                pr_debug("OF:␣parent␣bus␣is␣%s␣(na=%d,
                    ␣ns=%d)␣on␣%s\n",
                     pbus->name, pna, pns,
                        of_node_full_name(parent));
                /* Apply bus translation */
                //en nuestro caso no tenemos la
                    propiedad "rprop" por lo que
                    devuelve 1
                if (of_translate_one(dev, bus, pbus,
                    addr, na, ns, pna, rprop))
                        break;
                /* Complete the move up one level */
                na = pna;
                ns = pns;
                bus = pbus;
                of_dump_addr("OF:␣one␣level␣
                    translation:", addr, na);
        }
 bail:
        of_node_put(parent);
        of_node_put(dev);

        return result; }
-------------------------------------------------------------------------

static struct of_bus of_busses[] = {
#ifdef CONFIG_OF_ADDRESS_PCI
        /* PCI */
        {
                .name = "pci",
                .addresses = "assigned-addresses",
                .match = of_bus_pci_match,
```

```
                .count_cells = of_bus_pci_count_cells,
                .map = of_bus_pci_map,
                .translate = of_bus_pci_translate,
                .get_flags = of_bus_pci_get_flags,
        },
#endif /* CONFIG_OF_ADDRESS_PCI */
        /* ISA */
        {
                .name = "isa",
                .addresses = "reg",
                .match = of_bus_isa_match,
                .count_cells = of_bus_isa_count_cells,
                .map = of_bus_isa_map,
                .translate = of_bus_isa_translate,
                .get_flags = of_bus_isa_get_flags,
        },
        /* Default */
        {
                .name = "default",
                .addresses = "reg",
                .match = NULL,
                .count_cells =
                   of_bus_default_count_cells,
                .map = of_bus_default_map,
                .translate = of_bus_default_translate,
                .get_flags = of_bus_default_get_flags,
        },
};


static struct of_bus *of_match_bus(struct device_node
   *np) {
        int i;
        for (i = 0; i < ARRAY_SIZE(of_busses); i++)
                if (!of_busses[i].match || of_busses[i
                   ].match(np))
                        return &of_busses[i];
        BUG();
        return NULL;
}

/*  * Default translator (generic bus)  */
static void of_bus_default_count_cells(struct
   device_node *dev,
                                       int *addrc, int
                                           *sizec) {
```

```
        if (addrc)
                *addrc = of_n_addr_cells(dev); //
                    valor de la propiedad #address-
                    cells del dts busca desde el hijo
                    hasta el nodo raiz
        if (sizec)
                *sizec = of_n_size_cells(dev); //
                    valor de la propiedad #size-cells
                    del dts
}
```

Registro en sysfs

```
/**
 * gpiochip_add() - register a gpio_chip
 * @chip: the chip to register, with chip->base
    initialized
 * Context: potentially before irqs will work
 *
 * Returns a negative errno if the chip can't be
    registered, such as
 * because the chip->base is invalid or already
    associated with a
 * different chip.  Otherwise it returns zero as a
    success code.
 *
 * When gpiochip_add() is called very early during
    boot, so that GPIOs
 * can be freely used, the chip->dev device must be
    registered before
 * the gpio framework's arch_initcall().  Otherwise
    sysfs initialization
 * for GPIOs will fail rudely.
 *
 * If chip->base is negative, this requests dynamic
    assignment of
 * a range of valid GPIOs.
 */
int gpiochip_add(struct gpio_chip *chip) {
        unsigned long   flags;
        int             status = 0;
        unsigned        id;
        //la base es -1
        int             base = chip->base;
        struct gpio_desc *descs;
        //inizializa memoria para un array de chip->
           ngpio elementos
        //cada uno de tamaño struct gpio_desc
        //inizializa con 0 en sus contenidos
        descs = kcalloc(chip->ngpio, sizeof(descs[0]),
            GFP_KERNEL);
        if (!descs)
                return -ENOMEM;
        spin_lock_irqsave(&gpio_lock, flags);
        if (base < 0) {
                base = gpiochip_find_base(chip->ngpio)
```

```
                ;
        if (base < 0) {
                status = base;
                spin_unlock_irqrestore(&
                    gpio_lock, flags);
                goto err_free_descs;
        }
        chip->base = base;
}
status = gpiochip_add_to_list(chip);
if (status) {
        spin_unlock_irqrestore(&gpio_lock,
            flags);
        goto err_free_descs;
}
//valor inicial para la direccion de cada pin
    del GPIO
/*
* en la version del kernel v4.11.3
*
for (i = 0; i < chip->ngpio; i++) {
        struct gpio_desc *desc = &gdev->descs[
            i];
        desc->gdev = gdev;
        if (chip->get_direction) {
                int dir = chip->get_direction(
                    chip, i);
                if (!dir)
                        set_bit(FLAG_IS_OUT, &
                            desc->flags);
        } else if (!chip->direction_input) {
                set_bit(FLAG_IS_OUT, &desc->
                    flags);
        }
}
*/
for (id = 0; id < chip->ngpio; id++) {
        struct gpio_desc *desc = &descs[id];
        desc->chip = chip;
        /* REVISIT: most hardware initializes
            GPIOs as inputs (often
         * with pullups enabled) so power
             usage is minimized. Linux
         * code should set the gpio direction
             first thing; but until
         * it does, and in case chip->
```

```
                        get_direction is not set, we may
                     * expose the wrong direction in sysfs
                        .
                     */
                    desc->flags = !chip->direction_input ?
                        (1 << FLAG_IS_OUT) : 0;
            }
            chip->desc = descs;
            spin_unlock_irqrestore(&gpio_lock, flags);

    #ifdef CONFIG_PINCTRL
            INIT_LIST_HEAD(&chip->pin_ranges);
    #endif
            if (!chip->owner && chip->dev && chip->dev->
               driver)
                    chip->owner = chip->dev->driver->owner
                       ;
            status = gpiochip_set_desc_names(chip);
            if (status)
                    goto err_remove_from_list;
            status = of_gpiochip_add(chip);
            if (status)
                    goto err_remove_chip;
            acpi_gpiochip_add(chip);
            status = gpiochip_sysfs_register(chip);
            if (status)
                    goto err_remove_chip;
            pr_debug("%s:␣registered␣GPIOs␣%d␣to␣%d␣on␣
               device:␣%s\n", __func__,
                    chip->base, chip->base + chip->ngpio -
                        1,
                    chip->label ? : "generic");
            return 0;

    err_remove_chip:
            acpi_gpiochip_remove(chip);
            gpiochip_free_hogs(chip);
            of_gpiochip_remove(chip);
    err_remove_from_list:
            spin_lock_irqsave(&gpio_lock, flags);
            list_del(&chip->list);
            spin_unlock_irqrestore(&gpio_lock, flags);
            chip->desc = NULL;
    err_free_descs:
            kfree(descs);
```

```
        /* failures here can mean systems won't boot
           ... */
        pr_err("%s: GPIOs %d..%d (%s) failed to
           register\n", __func__,
             chip->base, chip->base + chip->ngpio -
                 1,
             chip->label ? : "generic");
        return status;
}
EXPORT_SYMBOL_GPL(gpiochip_add);

-----------------------------------------------------------------------


static int gpiochip_find_base(int ngpio) {
        struct gpio_chip *chip;
        //ARCH_NR_GPIOS = 512 por defecto
        // base inicial = 512-8 = 504
        int base = ARCH_NR_GPIOS - ngpio;
        /**
                 * list_for_each_entry_reverse -
                    iterate backwards over list of
                    given type.
                 * @pos:        the type * to use as a
                     loop cursor. // chip struct
                    gpio_chip
                * @head:       the head for your list
                   . //gpio_chips list_head
                * @member:     the name of the
                   list_head within the struct.
                recorre los existentes GPIO, por lo
                   tanto en GPIO0 no se itera y su
                   base es 504.
                veamos para GPIO1
         */
        list_for_each_entry_reverse(chip, &gpio_chips,
            list) {
                /* found a free space? para GPIO1*/
                //chip es GPIO0, 504+8 <= 504, es
                   falso
                if (chip->base + chip->ngpio <= base)
                        break;
                else
                        /* nope, check the space right
                            before the chip */
                        //GPIO 1 base = baseGPIO0 - 8
```

```
                                = 496
                        base = chip->base - ngpio;
        }
        if (gpio_is_valid(base)) {
                pr_debug("%s: found new base at %d\n",
                    __func__, base);
                return base;
        } else {
                pr_err("%s: cannot find free range\n",
                    __func__);
                return -ENOSPC;
        }
}

-------------------------------------------------------------------------------


/*
 * Add a new chip to the global chips list, keeping
    the list of chips sorted
 * by base order.
 *
 * Return -EBUSY if the new chip overlaps with some
    other chip's integer
 * space.
 */
static int gpiochip_add_to_list(struct gpio_chip *chip
   ) {
        struct list_head *pos;
        struct gpio_chip *_chip;
        int err = 0;
        /* find where to insert our chip */
        //gpio_chips es una list_head, linked list con
            todos los GPIO del sistema.
        //veremos el ejemplo con GPIO1
        //gpio_chips = [&gpio_chips,GPIO0,&gpio_chips]
        list_for_each(pos, &gpio_chips) {
                //_chip es GPIO0
                _chip = list_entry(pos, struct
                    gpio_chip, list);
                /* shall we insert before _chip? */
                //GPIO0 base = 504
                //GPIO1 base = 496
                //504 >= 496+8 es cierto
                if (_chip->base >= chip->base + chip->
                    ngpio)
```

```
                        break;
                }
                /* are we stepping on the chip right before?
                    */
                //pos != &gpio_chips es si hemos llegado al
                    final
                //pos->prev != &gpio_chips indica que la lista
                     no es vacia
                //si next y prev == &gpio_chips significa que
                    la lista es vacia
                //pos = GPIO0, pos->prev = &gpio_chips
                if (pos != &gpio_chips && pos->prev != &
                    gpio_chips) {
                        _chip = list_entry(pos->prev, struct
                            gpio_chip, list);
                        if (_chip->base + _chip->ngpio > chip
                            ->base) {
                                dev_err(chip->dev,
                                        "GPIO␣integer␣space␣
                                            overlap,␣cannot␣add␣
                                            chip\n");
                                err = -EBUSY;
                        }
                }
                if (!err)
                        //gpio_chips = [&gpio_chips,GPIO1,
                            GPIO0,&gpio_chips]
                        list_add_tail(&chip->list, pos);
                return err;
}

------------------------------------------------------------------------


//registramos el GPIO en sysfs
int gpiochip_sysfs_register(struct gpio_chip *chip) {
        struct device   *dev;
        /*
         * Many systems add gpio chips for SOC support
             very early,
         * before driver model support is available.
             In those cases we
         * register later, in gpiolib_sysfs_init() ...
             here we just
         * verify that _some_ field of gpio_class got
             initialized.
```

```
         */
        if (!gpio_class.p)
                return 0;
        /* use chip->base for the ID; it's already
           known to be unique */
        dev = device_create_with_groups(&gpio_class,
           chip->dev, MKDEV(0, 0),
                                        chip,
                                           gpiochip_groups
                                           ,
                                        "gpiochip%d",
                                           chip->base)
                                           ;
        if (IS_ERR(dev))
                return PTR_ERR(dev);
        mutex_lock(&sysfs_lock);
        chip->cdev = dev;
        mutex_unlock(&sysfs_lock);
        return 0;
}
```

Part V
System Calls mediante sysfs

UTILIDADES

```
/*
 * /sys/class/gpio/gpiochipN/
 *   /base ... matching gpio_chip.base (N)
 *   /label ... matching gpio_chip.label
 *   /ngpio ... matching gpio_chip.ngpio
 */
static ssize_t base_show(struct device *dev,
                                struct device_attribute
                                    *attr, char *buf) {
        const struct gpio_chip  *chip =
           dev_get_drvdata(dev);
        return sprintf(buf, "%d\n", chip->base);
}
static DEVICE_ATTR_RO(base);

static ssize_t label_show(struct device *dev,
                                struct device_attribute
                                    *attr, char *buf) {
        const struct gpio_chip  *chip =
           dev_get_drvdata(dev);
        return sprintf(buf, "%s\n", chip->label ? : ""
           );
}
static DEVICE_ATTR_RO(label);

static ssize_t ngpio_show(struct device *dev,
                                struct device_attribute
                                    *attr, char *buf) {
        const struct gpio_chip  *chip =
           dev_get_drvdata(dev);
        return sprintf(buf, "%u\n", chip->ngpio);
}
static DEVICE_ATTR_RO(ngpio);
```

EXPORT

```
/*
 * /sys/class/gpio/export ... write-only
 *      integer N ... number of GPIO to export (full
    access)
 * /sys/class/gpio/unexport ... write-only
 *      integer N ... number of GPIO to unexport
 */
static ssize_t export_store(struct class *class,
                                struct class_attribute
                                    *attr,
                                const char *buf,
                                   size_t len) {
        long                    gpio;
        struct gpio_desc        *desc;
        int                     status;

        //string to int
        status = kstrtol(buf, 0, &gpio);
        if (status < 0)
                goto done;

        desc = gpio_to_desc(gpio);
        /* reject invalid GPIOs */
        if (!desc) {
                pr_warn("%s: invalid GPIO %ld\n",
                   __func__, gpio);
                return -EINVAL;
        }
        /* No extra locking here; FLAG_SYSFS just
           signifies that the
         * request and export were done by on behalf
            of userspace, so
         * they may be undone on its behalf too.
         */
        status = gpiod_request(desc, "sysfs");
        if (status < 0) {
                if (status == -EPROBE_DEFER)
                        status = -ENODEV;
                goto done;
        }
        status = gpiod_export(desc, true);
        if (status < 0)
                gpiod_free(desc);
```

```
        else
                set_bit(FLAG_SYSFS, &desc->flags);

done:
        if (status)
                pr_debug("%s: status %d\n", __func__,
                    status);
        return status ? : len;
}

-----------------------------------------------

struct gpio_desc *gpio_to_desc(unsigned gpio) {
        struct gpio_chip *chip;
        unsigned long flags;

        spin_lock_irqsave(&gpio_lock, flags);
        //GPIO0 pin 3 = 504 + 3 = 507
        list_for_each_entry(chip, &gpio_chips, list) {
                //GPIO0 base = 504 gpio = 507 ngpio =
                    8
                if (chip->base <= gpio && chip->base +
                     chip->ngpio > gpio) {
                        spin_unlock_irqrestore(&
                            gpio_lock, flags);
                        return &chip->desc[gpio - chip
                            ->base];
                }
        }
        spin_unlock_irqrestore(&gpio_lock, flags);
        if (!gpio_is_valid(gpio))
                WARN(1, "invalid GPIO %d\n", gpio);
        return NULL;
}

int gpiod_export(struct gpio_desc *desc, bool
    direction_may_change){
        struct gpio_chip        *chip;
        struct gpiod_data       *data;
        [...]
        chip = desc->chip;
        [...]
        data = kzalloc(sizeof(*data), GFP_KERNEL);
        data->desc = desc;
        mutex_init(&data->mutex);
        // si en nuestro driver hemos creado funciones
```

```
        para direction_input y direction_output
    // estamos estableciendo que el GPIO puede
        cambiar su direccion y ser I/O
    if (chip->direction_input && chip->
        direction_output)
            data->direction_can_change =
                direction_may_change;
    else
            data->direction_can_change = false;

    [...]
    //creamos el gpio en /sys con gpio%d
        descriptor, GPIO0 3 = gpio507
    dev = device_create_with_groups(&gpio_class,
        chip->dev,
                                    MKDEV(0, 0),
                                        data,
                                        gpio_groups
                                        ,
                                    ioname ?
                                        ioname : "
                                        gpio%u",
                                    desc_to_gpio(
                                        desc));
}
```

DIRECCION

```
int gpiod_get_direction(struct gpio_desc *desc) {
        struct gpio_chip        *chip;
        unsigned                offset;
        int                     status = -EINVAL;

        chip = gpiod_to_chip(desc);
        offset = gpio_chip_hwgpio(desc);
        if (!chip->get_direction)
                return status;

        //vemos la definicion de get_direction
           implementada por nuestro driver
        status = chip->get_direction(chip, offset);
        if (status > 0) {
                /* GPIOF_DIR_IN, or other positive */
                status = 1;
                clear_bit(FLAG_IS_OUT, &desc->flags);
        }
        if (status == 0) {
                /* GPIOF_DIR_OUT */
                set_bit(FLAG_IS_OUT, &desc->flags);
        }
        return status;
}


static ssize_t direction_show(struct device *dev,
                struct device_attribute *attr, char *
                   buf) {
        struct gpiod_data *data = dev_get_drvdata(dev)
           ;
        struct gpio_desc *desc = data->desc;
        ssize_t                 status;

        mutex_lock(&data->mutex);
        gpiod_get_direction(desc);
        status = sprintf(buf, "%s\n",
                        test_bit(FLAG_IS_OUT, &desc->
                           flags)? "out" : "in");
        mutex_unlock(&data->mutex);
        return status;
}
```

```
-------------------------------------------------------------------------------

/* * gpiod_direction_input - set the GPIO direction to
     input */
int gpiod_direction_input(struct gpio_desc *desc) {
        struct gpio_chip        *chip;
        int                     status = -EINVAL;

        if (!desc || !desc->chip) {
                pr_warn("%s: invalid GPIO\n", __func__
                    );
                return -EINVAL;
        }

        chip = desc->chip;
        // se debe haber implementado en el driver el
           getValue y el direction_input
        if (!chip->get || !chip->direction_input) {
                gpiod_warn(desc,
                        "%s: missing get() or
                           direction_input()
                           operations\n",
                        __func__);
                return -EIO;
        }
        //hacemos la preparacion en nuestro driver
        status = chip->direction_input(chip,
           gpio_chip_hwgpio(desc));
        if (status == 0)
                clear_bit(FLAG_IS_OUT, &desc->flags);
                   //se cambia el valor en el GPIO

        trace_gpio_direction(desc_to_gpio(desc), 1,
           status);
        return status;
}

static int _gpiod_direction_output_raw(struct
   gpio_desc *desc, int value) {
        struct gpio_chip        *chip;
        int                     status = -EINVAL;

        /* GPIOs used for IRQs shall not be set as
           output */
        if (test_bit(FLAG_USED_AS_IRQ, &desc->flags))
           {
```

```
                gpiod_err(desc,
                          "%s: tried to set a GPIO
                             tied to an IRQ as output\
                             n",
                          __func__);
                return -EIO;
        }
        /* Open drain pin should not be driven to 1 */
        if (value && test_bit(FLAG_OPEN_DRAIN,  &desc
            ->flags))
                return gpiod_direction_input(desc);
        /* Open source pin should not be driven to 0
            */
        if (!value && test_bit(FLAG_OPEN_SOURCE,  &
            desc->flags))
                return gpiod_direction_input(desc);

        chip = desc->chip;
        // se debe haber implementado en el driver el
            setValue y el direction_output
        if (!chip->set || !chip->direction_output) {
                gpiod_warn(desc,
                       "%s: missing set() or
                          direction_output()
                          operations\n",
                       __func__);
                return -EIO;
        }
        status = chip->direction_output(chip,
            gpio_chip_hwgpio(desc), value);
        if (status == 0)
                set_bit(FLAG_IS_OUT, &desc->flags);
        trace_gpio_value(desc_to_gpio(desc), 0, value)
            ;
        trace_gpio_direction(desc_to_gpio(desc), 0,
            status);
        return status;
}


//se cambia la direccion y se inicializa su valor
static ssize_t direction_store(struct device *dev,
                struct device_attribute *attr, const
                    char *buf, size_t size) {
        struct gpiod_data *data = dev_get_drvdata(dev)
            ;
```

```
        struct gpio_desc *desc = data->desc;
        ssize_t                 status;

        mutex_lock(&data->mutex);
        if (sysfs_streq(buf, "high"))
                status = gpiod_direction_output_raw(
                   desc, 1);
        else if (sysfs_streq(buf, "out") ||
           sysfs_streq(buf, "low"))
                status = gpiod_direction_output_raw(
                   desc, 0);
        else if (sysfs_streq(buf, "in"))
                status = gpiod_direction_input(desc);
        else
                status = -EINVAL;
        mutex_unlock(&data->mutex);
        return status ? : size;
}
```

VALOR

```
//llamada del sysfs
static ssize_t value_show(struct device *dev,
                struct device_attribute *attr, char *
                   buf) {
        struct gpiod_data *data = dev_get_drvdata(dev)
           ;
        struct gpio_desc *desc = data->desc;
        ssize_t                 status;

        mutex_lock(&data->mutex);
        status = sprintf(buf, "%d\n",
           gpiod_get_value_cansleep(desc));
        mutex_unlock(&data->mutex);
        return status;
}

int gpiod_get_value_cansleep(const struct gpio_desc *
   desc) {
        int value;

        might_sleep_if(extra_checks);
        if (!desc)
                return 0;

        value = _gpiod_get_raw_value(desc);
        if (value < 0)
                return value;
        if (test_bit(FLAG_ACTIVE_LOW, &desc->flags))
                value = !value;
        return value;
}

static int _gpiod_get_raw_value(const struct gpio_desc
    *desc) {
        struct gpio_chip        *chip;
        int offset;     int value;
        chip = desc->chip;
        offset = gpio_chip_hwgpio(desc);
        //llamamos a la funcion get de nuestro driver
        value = chip->get ? chip->get(chip, offset) :
           -EIO;
        /*
         * FIXME: fix all drivers to clamp to [0,1] or
```

```
             return negative,
             * then change this to:
             * value = value < 0 ? value : !!value;
             * so we can properly propagate error codes.
             */
            value = !!value;
            trace_gpio_value(desc_to_gpio(desc), 1, value)
                ;
            return value;
}

------------------------------------------------------------------


static ssize_t value_store(struct device *dev,
                    struct device_attribute *attr, const
                        char *buf, size_t size) {
            struct gpiod_data *data = dev_get_drvdata(dev)
                ;
            struct gpio_desc *desc = data->desc;
            ssize_t                 status;

            mutex_lock(&data->mutex);
            //comprobar que el GPIO sea de salida
            if (!test_bit(FLAG_IS_OUT, &desc->flags)) {
                    status = -EPERM;
            } else {
                    long            value;
                    status = kstrtol(buf, 0, &value);
                    if (status == 0) {
                            gpiod_set_value_cansleep(desc,
                                value);
                            status = size;
                    }
            }
            mutex_unlock(&data->mutex);
            return status;
}

void gpiod_set_value_cansleep(struct gpio_desc *desc,
    int value) {
            might_sleep_if(extra_checks);
            if (!desc)
                    return;
            if (test_bit(FLAG_ACTIVE_LOW, &desc->flags))
                    value = !value;
```

```
        _gpiod_set_raw_value(desc, value);
}

static void _gpiod_set_raw_value(struct gpio_desc *
    desc, bool value) {
        struct gpio_chip        *chip;
        chip = desc->chip;
        trace_gpio_value(desc_to_gpio(desc), 0, value)
            ;

        if (test_bit(FLAG_OPEN_DRAIN, &desc->flags))
                _gpio_set_open_drain_value(desc, value
                    );
        else if (test_bit(FLAG_OPEN_SOURCE, &desc->
            flags))
                _gpio_set_open_source_value(desc,
                    value);
        else
                chip->set(chip, gpio_chip_hwgpio(desc)
                    , value);//por defecto llama a la
                    funcion set de nuestro driver
}
```