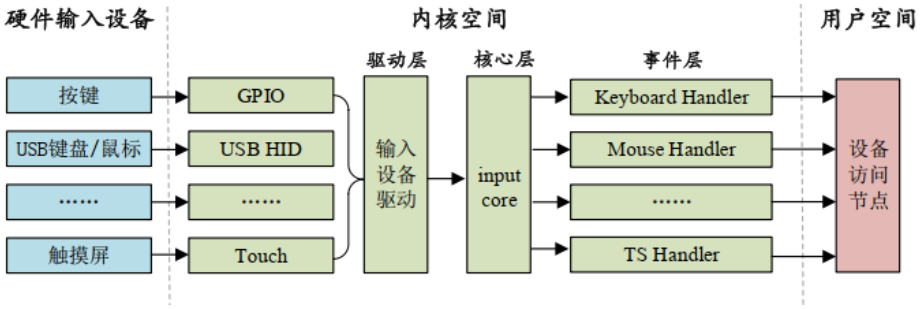
[L26]Linux Input 子系统

Linux内置很多常用的驱动框架,但这里有一种特殊的驱动框架,因此Linux把它独立出来,称为Input子系统,我们外部的硬件设备经过我们的驱动接收,通知到内核底层,然后变成标准的设备.



看看内核中input.h是怎么描述这个设备结构的.

```
struct input_dev {
  const char *name;
  const char *phys;
  const char *uniq;
  struct input_id id;

  unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)]; /* 设备能力(按位),在include/uapi/linux/input-event-codes.h定义,以INPUT_PROP_开头. */

  unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; /* 事件类型(按位),在include/uapi/linux/input-event-codes.h定义,以EV_开头. */
  unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; /* 按键值(按位),类比上面的,以KEY_开头. */
  unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; /* 相对坐标(按位),类比上面的. */
  unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; /* 绝对坐标(按位),类比上面的. */
  unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)]; /* 杂项事件(按位),类比上面的. */
  unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; /* LED设置(按位),类比上面的. */
  unsigned long sndbit[BITS_TO_LONGS(SND_CNT)]; /* 声音相关(按位),类比上面的. */
  unsigned long ffbit[BITS_TO_LONGS(FF_CNT)]; /* 压力反馈(按位),类比上面的. */
  unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; /* 开关状态(按位),类比上面的. */

  unsigned int hint_events_per_packet;

  unsigned int keycodemax;
  unsigned int keycodesize;
  void *keycode;

  int (*setkeycode)(struct input_dev *dev,
        const struct input_keymap_entry *ke,
        unsigned int *old_keycode);
  int (*getkeycode)(struct input_dev *dev,
        struct input_keymap_entry *ke);

  struct ff_device *ff;

  unsigned int repeat_key;
  struct timer_list timer;

  int rep[REP_CNT];

  struct input_mt *mt;

  struct input_absinfo *absinfo;

  unsigned long key[BITS_TO_LONGS(KEY_CNT)];
  unsigned long led[BITS_TO_LONGS(LED_CNT)];
  unsigned long snd[BITS_TO_LONGS(SND_CNT)];
  unsigned long sw[BITS_TO_LONGS(SW_CNT)];

  int (*open)(struct input_dev *dev);
  void (*close)(struct input_dev *dev);
  int (*flush)(struct input_dev *dev, struct file *file);
  int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
```

```
  struct input_handle __rcu *grab;

  spinlock_t event_lock;
  struct mutex mutex;

  unsigned int users;
  bool going_away;

  struct device dev;

  struct list_head  h_list;
  struct list_head  node;

  unsigned int num_vals;
  unsigned int max_vals;
  struct input_value *vals;

  bool devres_managed;
};

struct input_dev __must_check *input_allocate_device(void); /* 用来申请上面那块结构体. */
void input_free_device(struct input_dev *dev); /* 不用了要归还 */

int __must_check input_register_device(struct input_dev *); /* 和上节misc class一样效果的东西. */
void input_unregister_device(struct input_dev *);
```

那既然是一个输入设备,那么我们一定要接入一个输入的数据源,比如之前的dts已经写了一个按键,并且可以上访中断的,当然,还要有方法告诉内核才行啊,下面所有函数不管那种方法,最后都是到input_event然后告诉用户,我们知道有这么一个环节就行.

```
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);

// 举例

input_event(dev,EV_KEY,KEY_0,1); // 按下按键.

// 也可以写成

input_report_key(dev,KEY_0,1); // 按下按键

// 当然按键记得要松开.

input_report_key(dev,KEY_0,0);

// 最后还要记得把数据sync到系统.

input_sync(dev);

// 这句话效果也是一样的.

input_event(dev, EV_SYN, SYN_REPORT, 0);
```

设备的能力之类也是要预先设定的,也有多种方法,设定了能力就是告诉系统,我有能力上报哪些数据.

```
// 方法1

__set_bit(EV_KEY,inputdev->evbit); /* 能按按键 */
__set_bit(EV_REP,inputdev->evbit); /* 还能连按 */
__set_bit(KEY_0,inputdev->keybit); /* 能按KEY_0这个按键. */

// 方法2

inputdev->evbit[0] = BITMASK(EV_KEY) | BITMASK(EV_REP);
inputdev->keybit[BIT_WORD(KEY_0)] |= BIT_MASK(KEY_0);

// 方法3

keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);
input_set_capability(keyinputdev.inputdev, EV_KEY, KEY_0);
```

套用之前的方法,源码如下:

```
#include <linux/module.h>
#include <linux/of_gpio.h>
#include <linux/platform_device.h>
#include <linux/input.h>
#include <linux/timer.h>
#include <linux/of_irq.h>
#include <linux/interrupt.h>

struct key_dev
{
    struct input_dev *idev;
    struct timer_list timer;
    int gpio; /* gpio编号 */
    int irq;
};

static struct key_dev dev;

static void key_timer_function(unsigned long arg)
{
    /* 发送1就是按下,发送0就是抬起,这里模拟按下+抬起. */
    input_report_key(dev.idev, KEY_0, 1);
    input_sync(dev.idev);

    input_report_key(dev.idev, KEY_0, 0);
    input_sync(dev.idev);

    enable_irq(dev.irq);
}

static irqreturn_t key_interrupt(int irq, void *arg)
{
    /* 更严谨判断一下中断.实际上只有自己,如果要接入多个按键,可以共用中断. */
    if (dev.irq != irq)
        return IRQ_NONE;

    disable_irq_nosync(irq); /* 屏蔽按键中断 */

    /* 按键防抖处理，开启定时器延时15ms. */
    mod_timer(&dev.timer, jiffies + msecs_to_jiffies(15));
    return IRQ_HANDLED;
}

static int ps_key_init(struct platform_device *mdev)
{
    int ret;
    unsigned long irq_flags;

    dev.gpio = of_get_named_gpio(mdev->dev.of_node, "key-gpio", 0);
    if (!gpio_is_valid(dev.gpio))
    {
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio, "taterli-kernel-key");
    if (ret < 0)
    {
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    /* 将GPIO设置为输入模式 */
    gpio_direction_input(dev.gpio);

    dev.irq = irq_of_parse_and_map(mdev->dev.of_node, 0);
    if (!dev.irq)
    {
        return -EINVAL;
    }

    /* 获取设备树中指定的中断触发类型 */
    irq_flags = irq_get_trigger_type(dev.irq);
    if (IRQF_TRIGGER_NONE == irq_flags)
        irq_flags = IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING;
```

```
    printk("irq_num = %d,irq_flags = %ld\n", dev.irq, irq_flags);

    /* 申请中断 */
    ret = request_irq(dev.irq, key_interrupt, irq_flags, "PS EMIO IRQ", NULL);

    if (ret)
    {
        /* 实在没什么好办法做goto */
        gpio_free(dev.gpio);
        return ret;
    }

    /* 初始化定时器(要在中断之前做好!) */
    init_timer(&dev.timer);
    dev.timer.function = key_timer_function;

    /* 还记得之前LED的吗,类似套用就是. */

    /* 为dev指针分配内存 */
    dev.idev = devm_kzalloc(&mdev->dev, sizeof(struct key_dev), GFP_KERNEL);
    if (!dev.idev)
    {
        gpio_free(dev.gpio);
        return -ENOMEM;
    }

    platform_set_drvdata(mdev, dev.idev);

    dev.idev = devm_input_allocate_device(&mdev->dev);
    if (!dev.idev)
    {
        gpio_free(dev.gpio);
        return -ENOMEM;
    }

    dev.idev->name = "taterli-key";

    __set_bit(EV_KEY, dev.idev->evbit); /* 能按按键 */
    __set_bit(EV_REP, dev.idev->evbit); /* 还能连按 */
    __set_bit(KEY_0, dev.idev->keybit); /* 能按KEY_0这个按键. */

    return input_register_device(dev.idev);
}

static int ps_key_exit(struct platform_device *mdev)
{
    del_timer_sync(&dev.timer);

    input_unregister_device(dev.idev);

    gpio_free(dev.gpio);

    return 0;
}

/* 匹配列表 */
static const struct of_device_id key_of_match[] = {
    {.compatible = "taterli,key"},
    {}};

static struct platform_driver key_driver = {
    .driver = {
        .name = "taterl-key", /* 即使不用也要保留一个!  */
        .of_match_table = key_of_match,
    },
    .probe = ps_key_init,
    .remove = ps_key_exit,
};

static int __init key_driver_init(void)
{
    return platform_driver_register(&key_driver);
}

static void __exit key_driver_exit(void)
{
    platform_driver_unregister(&key_driver);
}
```

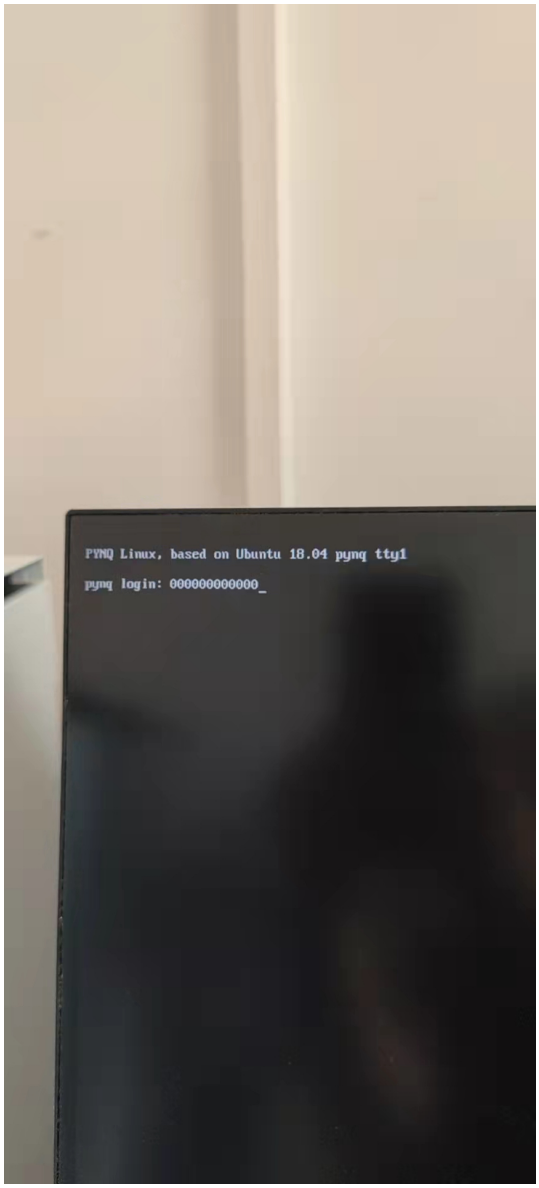
```
module_init(key_driver_init);
module_exit(key_driver_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Key GPIO");
MODULE_LICENSE("GPL");
```

测试可以用evtest工具,如果是PYNQ这些apt一下就能装.

```
xilinx@pynq:~$ sudo evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      taterli-key
Select the device event number [0-0]: 0
Input driver version is 1.0.1
Input device ID: bus 0x0 vendor 0x0 product 0x0 version 0x0
Input device name: "taterli-key"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 11 (KEY_0)
Key repeat handling:
  Repeat type 20 (EV_REP)
    Repeat code 0 (REP_DELAY)
      Value    250
    Repeat code 1 (REP_PERIOD)
      Value     33
Properties:
Testing ... (interrupt to exit)
Event: time 1647500289.935228, type 1 (EV_KEY), code 11 (KEY_0), value 1
Event: time 1647500289.935228, -------------- SYN_REPORT ------------
Event: time 1647500289.935248, type 1 (EV_KEY), code 11 (KEY_0), value 0
Event: time 1647500289.935248, -------------- SYN_REPORT ------------
Event: time 1647500290.555220, type 1 (EV_KEY), code 11 (KEY_0), value 1
Event: time 1647500290.555220, -------------- SYN_REPORT ------------
Event: time 1647500290.555235, type 1 (EV_KEY), code 11 (KEY_0), value 0
Event: time 1647500290.555235, -------------- SYN_REPORT ------------
Event: time 1647500291.085227, type 1 (EV_KEY), code 11 (KEY_0), value 1
Event: time 1647500291.085227, -------------- SYN_REPORT ------------
Event: time 1647500291.085244, type 1 (EV_KEY), code 11 (KEY_0), value 0
Event: time 1647500291.085244, -------------- SYN_REPORT ------------
Event: time 1647500291.205220, type 1 (EV_KEY), code 11 (KEY_0), value 1
Event: time 1647500291.205220, -------------- SYN_REPORT ------------
Event: time 1647500291.205235, type 1 (EV_KEY), code 11 (KEY_0), value 0
Event: time 1647500291.205235, -------------- SYN_REPORT ------------
```

当然他也确实个键盘,按一下0就打上去了.


PYNQ Linux, based on Ubuntu 18.04 pynq tty1
pynq login: 000000000000