**实验三. 按键中断驱动及控制实验**

1. 实验目的

 了解 ARM 设备外围电路结构与接口原理

 熟悉 Linux 系统下硬件中断驱动编程

 编程实现对嵌入式设备上按键的控制

2. 实验内容

 阅读 IMX6 平台硬件文档，熟悉 ARM 处理硬件外围接口电路

 编程实现 IMX6 平台设备上按键设备中断驱动控制及应用测试程序

3. 实验环境

 硬件：IMX6 教学平台，PC 机酷睿 i3 以上, 硬盘 120G 以上, 内存 2G 以上

 软件：Vmware Workstation +Yocto 项目

4. 实验原理

**4.1 硬件接口原理**

◆ IMX6 按键硬件接口

IMX6 平台上共有 12 个按键。KEY1~KEY12 按键分别接入到 IMX6 处理器的 12 个引脚。按键电路图 如下图所示：

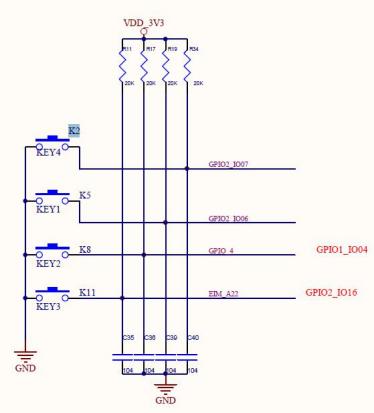


图 3.3. 1 按键电路原理图



◆ Linux input 子系统

Linux input子系统是 Linux 中为支持输入设备而设计的驱动程序。输入子系统在内核中由输入子系统的 事件处理层(Event Input Hander) 、输入子系统的核心层（Input Core）和输入子系统的设备驱动层组成。对于 Linux 驱动开发人员而言，在基于输入子系统的事件处理层和输入子系统的核心层编写具体的输入设备驱动 程序。输入子系统的一般工作机制是，输入设备动作时使内核产生中断，在中断过程中驱动将输入设备产 生的数据（比如触摸屏的坐标值）放到一个缓冲区中，并通知应用层，应用层通过 read 读取驱动层的数据。

下面是 Linux input 子系统的分层结构图：

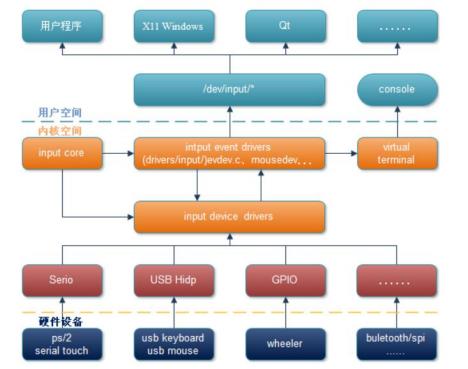


图 3.3.2 input 子系统分层结构图

下面介绍编写输入子系统具体的设备驱动程序需要关心的 API 和输入子系统比较重要的数据结构。 重要的 API 函数：

◆ 分配一个输入设备

struct input\_dev \*input\_allocate\_device\* (void);

◆ 释放一个输入设备

Void input\_free\_device\* ( struct input\_dev \*dev);

◆ 向 input core 层注册一个输入设备

int input\_register\_device (struct input\_dev \*dev);

◆ 向 input core 层注消一个输入设备

|  |  |  |  |
| --- | --- | --- | --- |
| int input unregister device (struct input dev \*dev); | | | |
| ◆ 报告指定 type,code 的输入事件 | | | |
| void input event (struct input dev \*dev,unsigned int type,unsigned int code,int | | | |
| value); | | | |
| dev ：指向 input device 的指针。 | | | |
| type：输入类型（ EV\_KEY、EV\_ABS 等）。 | | | |
| code：输入按键（例如 EV KEY 的 KEY 1）。 | | | |
| value：按鍵值 (按下或抬起) | | | |
| ◆ 报告键值 | | | |
| void input report key (struct input dev \*dev,unsigned int code,int value); | | | |
| ◆ 报告同步事件 | | | |
| void input sync (struct input dev \*dev); | | | |
| 定义输入设备类型（Event types）：  我们可以设定 struct input\_dev 里的 evbit 成员，来定义所要接受的输入类型，可用的输入类型如 下： | | | |
| #define | EV SYN | | 0x00 |
| #define | EV KEY | | 0x01 |
| #define | EV REL  \_\_\_\_\_\_\_\_\_\_ | | 0x02 |
| #define | EV ABS | | 0x03 |
| #define | EV MSC | | 0x04 |
| #define | EV SW | | 0x05 |
| #define | EV LED  \_ | | 0x11 |
| #define | EV SND  \_\_ | | 0x12 |
| #define | EV REP | | 0x14 |
| #define | EV FF  \_ | | 0x15 |
| #define | EV PWR  \_ | | 0x16 |
| #define | EV FF STATUS  \_\_ \_\_ | | 0x17 |
| #define | EV MAX | | 0x1f |
| #define | EV CNT  \_ | | (EV\_MAX+1) |
| 定义输入设备 code（Event codes）： 下面是键盘部分的 Event codes | | | |
| #define | KEY RESERVED | 0 | |
| #define | KEY ESC | 1 | |
| #define | KEY 1 | 2 | |
| #define | KEY 2 | 3 | |
| #define | KEY 3 | 4 | |
| #define | KEY 4 | 5 | |
| #define | KEY 5 | 6 | |
| #define | KEY 6 | 7 | |
| #define | KEY 7  \_ | 8 | |

#define KEY 8 9

#define KEY 9 10

… …

输入事件结构体：

struct input\_event {

struct timeval time;

u16 type; //Event types

u16 code; //Event codes

\_\_

s32 value; //Event value

\_\_

} ;

input\_dev 结构体，一个 input\_dev 结构代表了一个输入设备：

struct input\_dev {

const char \*name;//名称，cat /proc/input/devices 可以看到每个输入设备的信息，其中 name 就是这里指定的

const char \*phys;

const char \*uniq;

struct input\_id id;

|  |  |  |
| --- | --- | --- |
| unsigned | long | evbit [BITS\_TO\_LONGS (EV\_CNT)]; //支持的事件类型 |
| unsigned | long | keybit [BITS\_TO\_LONGS (KEY\_CNT)];//支持键盘事件 |
| unsigned | long | relbit [BITS\_TO\_LONGS (REL\_CNT)]; |
| unsigned | long | absbit [BITS\_TO\_LONGS (ABS\_CNT)]; |
| unsigned | long | mscbit [BITS\_TO\_LONGS (MSC\_CNT)]; |
| unsigned | long | ledbit [BITS\_TO\_LONGS (LED\_CNT)]; |
| unsigned | long | sndbit [BITS\_TO\_LONGS (SND\_CNT)]; |
| unsigned | long | ffbit [BITS\_TO\_LONGS (FF\_CNT)]; |
| unsigned | long | swbit [BITS\_TO\_LONGS (SW\_CNT)]; |
| } … … |  |  |

以上具体内容参见内核源码的 include/linux/input.h 文件。

**4.2 关键代码分析**

◆ 驱动程序分析：

/\*

\* Driver for keys on GPIO lines capable of generating interrupts .

\*

\* Copyright 2005 Phil Blundell

\* Copyright 2010, 2011 David Jander <david@protonic .nl>

\* Copyright (C) 2013 Freescale Semiconductor, 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 .

\*/

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/interrupt .h>

#include <linux/irq .h>

#include <linux/sched .h>

#include <linux/pm.h>

#include <linux/slab.h>

#include <linux/sysctl.h>

#include <linux/proc\_fs .h>

#include <linux/delay .h>

#include <linux/platform\_device .h>

#include <linux/input .h>

#include <linux/gpio\_keys .h>

#include <linux/workqueue .h>

#include <linux/gpio.h>

#include <linux/of.h>

#include <linux/of\_platform .h>

#include <linux/of\_gpio .h>

#include <linux/spinlock.h>

struct gpio\_button\_data {

const struct gpio\_keys\_button \*button;

struct input\_dev \*input;

struct timer list timer;

struct work struct work;

unsigned int timer\_debounce; /\* in msecs \*/

unsigned int irq;

spinlock\_t lock;

bool disabled;

bool key\_pressed;

};

struct gpio\_keys\_drvdata {

const struct gpio\_keys\_platform\_data \*pdata;

struct input\_dev \*input;

struct mutex disable lock;

\_

struct gpio\_button\_data data [0];

};

/\*

\* SYSFS interface for enabling/disabling keys and switches :

\*

\* There are 4 attributes under /sys/devices/platform/gpio-keys/

\* keys [ro] - bitmap of keys (EV\_KEY) which can be

\* disabled

\* switches [ro] - bitmap of switches (EV\_SW) which can be

\* disabled

\* disabled\_keys [rw] - bitmap of keys currently disabled

\* disabled switches [rw] - bitmap of switches currently disabled

\_

\*

\* Userland can change these values and hence disable event generation

\* for each key (or switch). Disabling a key means its interrupt line

\* is disabled .

\*

\* For example, if we have following switches set up as gpio-keys :

\* SW\_DOCK = 5

\* SW\_CAMERA\_LENS\_COVER = 9



\* SW\_KEYPAD\_SLIDE = 10

\* 11-9,5

\* Next we want to disable proximity (11) and dock (5), we write:

\* 11,5

\* to file disabled\_switches . Now proximity and dock IRQs are disabled.

\* This can be verified by reading the file disabled\_switches:

\* 11,5

\* If we now want to enable proximity (11) switch we write:

\* 5

\* to disabled switches.

\_

\*

\* We can disable only those keys which don 't allow sharing the irq .

\*/

/\*\*

\* get\_n\_events\_by\_type () - returns maximum number of events per @type

\* @type : type of button (%EV\_KEY, %EV\_SW)

\*

\* Return value of this function can be used to allocate bitmap

\* large enough to hold all bits for given type.

\*/

static inline int get\_n\_events\_by\_type (int type) {

BUG\_ON (type != EV\_SW && type != EV\_KEY);

return (type == EV\_KEY) ? KEY\_CNT : SW\_CNT;

}

/\*\*

\* gpio\_keys\_disable\_button () - disables given GPIO button

\* @bdata: button data for button to be disabled

\*

\* Disables button pointed by @bdata . This is done by masking

\* IRQ line. After this function is called, button won 't generate

\* input events anymore. Note that one can only disable buttons

\* that don 't share IRQs .

\*

\* Make sure that @bdata->disable\_lock is locked when entering

\* this function to avoid races when concurrent threads are

\* disabling buttons at the same time.

\*/

static void gpio\_keys\_disable\_button (struct gpio\_button\_data \*bdata)

{

if ( !bdata->disabled) {

/\*

\* Disable IRQ and possible debouncing timer .

\*/

disable\_irq(bdata->irq);

if (bdata->timer\_debounce)

del\_timer\_sync (&bdata->timer);

bdata->disabled = true;

}

}

/\*\*

\* SW\_FRONT\_PROXIMITY = 11

\* This is read from switches:



struct gpio\_button\_data \*bdata = &ddata->data [i];

bu f [ret++] = '\n ';

\* gpio\_keys\_enable\_button () - enables given GPIO button

\* @bdata: button data for button to be disabled \*

\* Enables given button pointed by @bdata. \*

\* Make sure that @bdata->disable\_lock is locked when entering

\* this function to avoid races with concurrent threads trying

\* to enable the same button at the same time.

\*/

static void gpio\_keys\_enable\_button (struct gpio\_button\_data \*bdata)

{

if (bdata->disabled) {

enable irq(bdata->irq);

bdata- = false;

}

}

/\*\*

\* gpio\_keys\_attr\_show\_helper () - fill in stringified bitmap of buttons

\* @ddata: pointer to drvdata

\* @bu f: buffer where stringified bitmap is written

\* @type : button type (%EV\_KEY, %EV\_SW)

\* @only\_disabled: does caller want only those buttons that are

\* currently disabled or all buttons that can be

\* disabled \*

\* This function writes buttons that can be disabled to @bu f. If

\* @only\_disabled is true, then @buf contains only those buttons

\* that are currently disabled . Returns 0 on success or negative

\* errno on failure.

\*/

static ssize\_t gpio\_keys\_attr\_show\_helper (struct gpio\_keys\_drvdata \*ddata,

char \*buf, unsigned int type,

bool only\_disabled)

{

int n\_events = get n events\_by\_type (type);

unsigned long \*bits;

ssize t ret;

int i;

bits = kcalloc (BITS\_TO\_LONGS (n\_events), sizeof (\*bits), GFP\_KERNEL);

if ( !bits)

return -ENOMEM;

for (i = 0; i < ddata->pdata->nbuttons; i++) {

if (bdata->button->type != type) continue;

if (only\_disabled && !bdata->disabled)

continue;

set\_bit (bdata->button->code, bits);

}

ret = bitmap\_scnlistprintf (buf, PAGE\_SIZE - 2, bits, n\_events);

bu f [ret] = '\0 '; kfree (bits);

return ret;

}

/\*\*

\* gpio\_keys\_attr\_store\_helper () - enable/disable buttons based on given bitmap

\* @ddata: pointer to drvdata

\* @bu f: buffer from userspace that contains stringified bitmap

\* @type : button type (%EV\_KEY, %EV\_SW)

\*

\* This function parses stringified bitmap from @buf and disables/enables

\* GPIO buttons accordingly. Returns 0 on success and negative error

\* on failure.

\*/

static ssize t gpio\_keys\_attr\_store\_helper (struct gpio\_keys\_drvdata \*ddata,

const char \*bu f, unsigned int type) {

int n\_events = get n events\_by\_type (type);

unsigned long \*bits;

ssize t error;

int i;

bits = kcalloc (BITS\_TO\_LONGS (n\_events), sizeof (\*bits), GFP\_KERNEL);

if ( !bits)

return -ENOMEM;

error = bitmap\_parselist (buf, bits, n\_events);

if (error)

goto out;

/\* First validate \*/

for (i = 0; i < ddata->pdata->nbuttons; i++) {

if (bdata->button->type != type) continue;

if (test\_bit (bdata->button->code, bits) &&

!bdata->button->can disable) { error = -EINVAL; \_

goto out;

}

}

mutex lock (&ddata->disable\_lock);

for (i = 0; i < ddata->pdata->nbuttons; i++) {

if (bdata->button->type != type) continue;

if (test\_bit (bdata->button->code, bits))

gpio\_keys\_disable\_button (bdata);

else

struct gpio\_button\_data \*bdata = &ddata->data [i];

struct gpio\_button\_data \*bdata = &ddata->data [i];

\_

\_

gpio\_keys\_enable\_button (bdata);

}

mutex\_unlock (&ddata->disable\_lock);

out:

kfree (bits);

return error;

}

#define ATTR\_SHOW\_FN (name, type, only\_disabled) \

static ssize\_t gpio\_keys\_show\_##name (struct device \*dev, \

struct device\_attribute \*attr, \

char \*buf) \

{ \

struct platform\_device \*pdev = to\_platform\_device (dev); \

struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata (pdev); \

\

return gpio\_keys\_attr\_show\_helper (ddata, bu f, \

type, only\_disabled); \

}

ATTR SHOW FN (keys, EV\_KEY, false);

\_ \_

ATTR\_SHOW\_FN (switches, EV\_SW, false);

ATTR\_SHOW\_FN (disabled\_keys, EV\_KEY, true);

ATTR\_SHOW\_FN (disabled\_switches, EV\_SW, true);

/\*

\* ATTRIBUTES: \*

\* /sys/devices/platform/gpio-keys/keys [ro]

\* /sys/devices/platform/gpio-keys/switches [ro]

\*/

static DEVICE\_ATTR(keys, S\_IRUGO, gpio\_keys\_show\_keys, NULL);

static DEVICE ATTR(switches, S\_IRUGO, gpio\_keys\_show\_switches, NULL);

#define ATTR\_STORE\_FN (name, type) \

static ssize\_t gpio\_keys\_store\_##name (struct device \*dev, \

struct device\_attribute \*attr, \

const char \*buf, \

size\_t count) \

{ \

struct platform\_device \*pdev = to\_platform\_device (dev); \

struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata (pdev); \

ssize t error; \

\

error = gpio\_keys\_attr\_store\_helper (ddata, bu f, type); \

if (error) \

return error; \

\

return count; \

}

ATTR\_STORE\_FN (disabled\_keys, EV\_KEY);

ATTR\_STORE\_FN (disabled\_switches, EV\_SW);

/\*

\* ATTRIBUTES:

\*

\* /sys/devices/platform/gpio-keys/disabled\_keys [rw]

\* /sys/devices/platform/gpio-keys/disables\_switches [rw]

\*/

static DEVICE ATTR(disabled\_keys, S\_IWUSR | S\_IRUGO,

\_

gpio\_keys\_show\_disabled\_keys,

gpio\_keys\_store\_disabled\_keys);

static DEVICE ATTR(disabled\_switches, S\_IWUSR | S\_IRUGO,

\_

gpio\_keys\_show\_disabled\_switches,

gpio\_keys\_store\_disabled\_switches);

static struct attribute \*gpio\_keys\_attrs [] = {

&dev attr keys .attr,

\_ \_

&dev attr switches .attr,

\_ \_

&dev attr disabled keys.attr,

\_ \_ \_

&dev attr disabled switches.attr,

\_ \_ \_

NULL,

};

static struct attribute\_group gpio\_keys\_attr\_group = {

.attrs = gpio\_keys\_attrs,

};

static void gpio\_keys\_gpio\_report\_event (struct gpio\_button\_data \*bdata)

{

const struct gpio\_keys\_button \*button = bdata->button;

struct input\_dev \*input = bdata->input;

unsigned int type = button->type ?: EV\_KEY; ^

int state = (gpio\_get\_value\_cansleep (button->gpio) ? 1 : 0)

button->active low;

\_

if (type == EV\_ABS) {

if (state)

input\_event (input, type, button->code, button->value);

} else {

input\_event (input, type, button->code, ! !state);

}

input\_sync (input);

}

static void gpio\_keys\_gpio\_work\_func (struct work\_struct \*work)

{

struct gpio\_button\_data \*bdata =

container of (work, struct gpio\_button\_data, work);

\_

gpio\_keys\_gpio\_report\_event (bdata);

if (bdata->button->wakeup)

pm\_relax (bdata->input->dev .parent);

}

static void gpio\_keys\_gpio\_timer (unsigned long \_data)

{

struct gpio\_button\_data \*bdata = (struct gpio\_button\_data \*)\_data;

schedule work (&bdata->work);

\_

}

static irqreturn\_t gpio\_keys\_gpio\_is r (int irq, void \*dev\_id) {

struct gpio\_button\_data \*bdata = dev\_id;

BUG\_ON (irq != bdata->irq);

if (bdata->button->wakeup)

pm\_stay\_awake (bdata->input->dev.parent);

if (bdata->timer\_debounce)

mod\_timer (&bdata->timer,

jiffies + msecs\_to\_jiffies (bdata->timer\_debounce));

else

schedule work (&bdata->work);

\_

return IRQ HANDLED;

}

static void gpio\_keys\_irq\_timer (unsigned long \_data) {

struct gpio\_button\_data \*bdata = (struct gpio\_button\_data \*)\_data;

struct input\_dev \*input = bdata->input;

unsigned long flags;

spin\_lock\_irqsave (&bdata->lock, flags);

if (bdata->key\_pressed) {

input\_event (input, EV\_KEY, bdata->button->code, 0);

input\_sync (input);

bdata->key\_pressed = false;

}

spin\_unlock\_irqrestore (&bdata->lock, flags);

}

static irqreturn\_t gpio\_keys\_irq\_is r (int irq, void \*dev\_id) {

struct gpio\_button\_data \*bdata = dev\_id;

const struct gpio\_keys\_button \*button = bdata->button;

struct input\_dev \*input = bdata->input;

unsigned long flags;

BUG\_ON (irq != bdata->irq);

spin\_lock\_irqsave (&bdata->lock, flags);

if ( !bdata->key\_pressed) {

if (bdata->button->wakeup)

pm\_wakeup\_event (bdata->input->dev.parent, 0);

input\_event (input, EV\_KEY, button->code, 1);

input\_sync (input);

if ( !bdata->timer\_debounce) {

input\_event (input, EV\_KEY, button->code, 0);

input\_sync (input);

goto out;

}

bdata->key\_pressed = true; }

if (bdata->timer\_debounce)

mod timer (&bdata->timer,

\_

jiffies + msecs\_to\_jiffies (bdata->timer\_debounce));

out:

spin\_unlock\_irqrestore (&bdata->lock, flags);

return IRQ HANDLED;

}

static int gpio\_keys\_setup\_key (struct platform\_device \*pdev,

struct input\_dev \*input,

struct gpio\_button\_data \*bdata,

const struct gpio\_keys\_button \*button)

{

const char \*desc = button->desc ? button->desc : "gpio\_keys";

struct device \*dev = &pdev->dev;

irq\_handler\_t isr;

unsigned long irqflags;

int irq, error;

bdata->input = input;

bdata->button = button;

spin\_lock\_init (&bdata->lock);

if (gpio\_is\_valid(button->gpio)) {

error = gpio\_request\_one (button->gpio, GPIOF\_IN, desc);

if (error < 0) {

dev err (dev, "Failed to request GPIO %d, error %d\n",

\_

button->gpio, error);

return error;

}

if (button->debounce\_interval) {

error = gpio\_set\_debounce (button->gpio,

button->debounce interval \* 1000);

/\* use timer if gpiolib doesn 't provide debounce \*/

if (error < 0)

bdata->timer\_debounce =

button->debounce interval;

}

irq = gpio\_to\_irq(button->gpio);

if (irq < 0) {

error = irq;

dev err (dev,

\_

"Unable to get irq number for GPIO %d, error %d\n",

button->gpio, error);

goto fail;

}

bdata->irq = irq;

INIT WORK (&bdata->work, gpio\_keys\_gpio\_work\_func);

\_

setup\_timer (&bdata->timer,

gpio\_keys\_gpio\_timer, (unsigned long)bdata);

is r = gpio\_keys\_gpio\_is r;

irqflags = IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING;

if (bdata->button->wakeup)

irqflags |= IRQF\_NO\_SUSPEND;

} else {

if ( !button->irq) {

dev\_err (dev, "No IRQ specified\n");

return -EINVAL; }

bdata->irq = button->irq;

if (button->type && button->type != EV\_KEY) {

dev\_err (dev, "Only EV\_KEY allowed for IRQ buttons.\n");

return -EINVAL;

}

bdata->timer\_debounce = button->debounce\_interval;

setup\_timer (&bdata->timer,

gpio\_keys\_irq\_timer, (unsigned long)bdata);

is r = gpio\_keys\_irq\_isr;

irqflags = 0;

}

input\_set\_capability (input, button->type ?: EV\_KEY, button->code);

/\*

\* If platform has specified that the button can be disabled,

\* we don 't want it to share the interrupt line.

\*/

if ( !button->can\_disable)

irqflags |= IRQF\_SHARED;

error = request\_any\_context\_irq(bdata->irq, is r, irqflags, desc, bdata); if (error < 0) {

dev err (dev, "Unable to claim irq %d; error %d\n",

bdata->irq, error);

goto fail;

}

return 0;

fail :

if (gpio\_is\_valid(button->gpio))

gpio\_free (button->gpio);

return error;

}

static void gpio\_keys\_report\_state (struct gpio\_keys\_drvdata \*ddata) {

struct input\_dev \*input = ddata->input; int i;

for (i = 0; i < ddata->pdata->nbuttons; i++) {

struct gpio\_button\_data \*bdata = &ddata->data [i];

if (gpio\_is\_valid(bdata->button->gpio))

gpio\_keys\_gpio\_report\_event (bdata);

}

input\_sync (input);

}

static int gpio\_keys\_open (struct input\_dev \*input)

{

struct gpio\_keys\_drvdata \*ddata = input\_get\_drvdata (input);

const struct gpio\_keys\_platform\_data \*pdata = ddata->pdata;

int error;

if (pdata->enable) {

error = pdata->enable (input->dev .parent);

if (error)

return error;

}

/\* Report current state of buttons that are connected to GPIOs \*/

gpio\_keys\_report\_state (ddata);

return 0;

}

static void gpio\_keys\_close (struct input\_dev \*input)

{

struct gpio\_keys\_drvdata \*ddata = input\_get\_drvdata (input);

const struct gpio\_keys\_platform\_data \*pdata = ddata->pdata;

if (pdata->disable)

pdata->disable (input->dev.parent);

}

/\*

\* Handlers for alternative sources of platform\_data

\*/

#ifdef CONFIG OF

/\*

\* Translate OpenFirmware node properties into platform\_data

\*/

static struct gpio\_keys\_platform\_data \*

gpio\_keys\_get\_devtree\_pdata (struct device \*dev)

{

struct device node \*node, \*pp;

\_

struct gpio\_keys\_platform\_data \*pdata;

struct gpio\_keys\_button \*button;

int error;

int nbuttons;

int i;

node = dev->of\_node;

if ( !node) {

error = -ENODEV;

goto err\_out;

}

nbuttons = of\_get\_child\_count (node);

if (nbuttons == 0) {

goto err\_out;

error = -ENODEV;

}

pdata = kzalloc (sizeof (\*pdata) + nbuttons \* (sizeof \*button),

GFP KERNEL);

\_

if ( !pdata) {

error = -ENOMEM;

goto err\_out;

}

pdata->buttons = (struct gpio\_keys\_button \*) (pdata + 1);

pdata->nbuttons = nbuttons;

pdata->rep = ! !of\_get\_property (node, "autorepeat", NULL);

i = 0;

for each child of node (node, pp) {

\_ \_ \_ \_

int gpio;

enum of gpio\_flags flags;

\_

if ( !of\_find\_property (pp, "gpios", NULL)) {

pdata->nbuttons--;

dev warn (dev, "Found button without gpios\n");

\_

continue;

}

gpio = of\_get\_gpio\_flags (pp, 0, &flags);

if (gpio < 0) {

error = gpio;

if (error != -EPROBE\_DEFER)

dev err (dev,

\_

"Failed to get gpio flags, error: %d\n",

error);

goto err\_free\_pdata;

}

button = &pdata->buttons [i++];

button->gpio = gpio;

button->active\_low = flags & OF\_GPIO\_ACTIVE\_LOW;

if (of\_property\_read\_u32 (pp, "linux,code", &button->code)) {

dev err (dev, "Button without keycode: 0x%x\n",

\_

button->gpio); error = -EINVAL;

goto err\_free\_pdata;

}

button->desc = of\_get\_property (pp, "label", NULL);

if (of\_property\_read\_u32 (pp, "linux,input-type", &button->type))

button->type = EV\_KEY;

button->wakeup = ! !of\_get\_property (pp, "gpio-key,wakeup", NULL);

if (of\_property\_read\_u32 (pp, "debounce-interval",

&button->debounce interval)) button->debounce\_interva = 5;

}

error = -EINVAL;

if (pdata->nbuttons == 0) {

goto err\_free\_pdata;

}

return pdata;

err free pdata :

\_ \_

kfree (pdata);

err out:

return ERR PTR (error);

\_

}

static struct of\_device\_id gpio\_keys\_of\_match [] = {

{ .compatible = "gpio-keys", },

{ },

};

MODULE DEVICE TABLE (of, gpio\_keys\_of\_match);

\_ \_

#else

static inline struct gpio\_keys\_platform\_data \*

gpio\_keys\_get\_devtree\_pdata (struct device \*dev)

{

return ERR PTR (-ENODEV);

}

#endif

static void gpio\_remove\_key (struct gpio\_button\_data \*bdata)

{

free irq(bdata->irq, bdata);

\_

if (bdata->timer\_debounce)

del timer sync (&bdata->timer);

\_ \_

cancel work sync (&bdata->work);

\_ \_

if (gpio\_is\_valid(bdata->button->gpio))

gpio\_free (bdata->button->gpio);

}

static int gpio\_keys\_probe (struct platform\_device \*pdev)

{

struct device \*dev = &pdev->dev;

const struct gpio\_keys\_platform\_data \*pdata = dev\_get\_platdata (dev);

struct gpio\_keys\_drvdata \*ddata;

struct input\_dev \*input;

int i, error;

int wakeup = 0;

if ( !pdata) {

pdata = gpio\_keys\_get\_devtree\_pdata (dev);

if (IS\_ERR(pdata))

return PTR ERR(pdata);

\_

}

ddata = kzalloc (sizeof (struct gpio\_keys\_drvdata) +

pdata->nbuttons \* sizeof (struct gpio\_button\_data),

GFP KERNEL);

\_

input = input\_allocate\_device ();

if ( !ddata || !input) {

dev err (dev, "failed to allocate state\n");

errr = -ENOMEM;

goto fail1;

}

ddata->pdata = pdata;

ddata->input = input;

mutex init (&ddata->disable\_lock);

\_

platform\_set\_drvdata (pdev, ddata);

input\_set\_drvdata (input, ddata);

/\* Enable auto repeat feature of Linux input subsystem \*/

input->name = pdata->name ? : pdev->name;

input->dev .parent = &pdev->dev;

input->close = gpio\_keys\_close;

input->id.bustype = BUS\_HOST;

input->id.product = 0x0001;

input->phys = "gpio-keys/input0";

input->open = gpio\_keys\_open;

input->id.version = 0x0100;

input->id.vendor = 0x0001;

if (pdata->rep)

set bit (EV\_REP, input->evbit);

\_\_ \_

for (i = 0; i < pdata->nbuttons; i++) {

const struct gpio\_keys\_button \*button = &pdata->buttons [i];

struct gpio\_button\_data \*bdata = &ddata->data [i];

error = gpio\_keys\_setup\_key (pdev, input, bdata, button);

if (error)

goto fail2;

if (button->wakeup) wakeup = 1;

}

error = sysfs\_create\_group (&pdev->dev.kobj, &gpio\_keys\_attr\_group);

if (error) {

dev err (dev, "Unable to export keys/switches, error: %d\n",

\_

error);

goto fail2;

}

error = input\_register\_device (input);

if (error) {

dev err (dev, "Unable to register input device, error: %d\n",

\_

error);

goto fail3;

}

device init wakeup (&pdev->dev, wakeup);

\_ \_



return 0;

fail3 :

sysfs\_remove\_group (&pdev->dev.kobj, &gpio\_keys\_attr\_group);

fail2 :

while (--i >= 0)

gpio\_remove\_key (&ddata->data [i]);

fail1 :

input\_free\_device (input);

kfree (ddata);

/\* If we have no platform data, we allocated pdata dynamically. \*/

if ( !dev\_get\_platdata (&pdev->dev))

kfree (pdata);

return error;

}

static int gpio\_keys\_remove (struct platform\_device \*pdev)

{

struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata (pdev);

struct input\_dev \*input = ddata->input;

int i;

sysfs\_remove\_group (&pdev->dev.kobj, &gpio\_keys\_attr\_group);

device init wakeup (&pdev->dev, 0);

\_ \_

for (i = 0; i < ddata->pdata->nbuttons; i++)

gpio\_remove\_key (&ddata->data [i]);

input\_unregister\_device (input);

/\* If we have no platform data, we allocated pdata dynamically. \*/

if ( !dev\_get\_platdata (&pdev->dev))

kfree (ddata->pdata);

kfree (ddata);

return 0;

}

#ifdef CONFIG PM SLEEP

static int gpio\_keys\_suspend (struct device \*dev)

{

struct gpio\_keys\_drvdata \*ddata = dev\_get\_drvdata (dev);

struct input\_dev \*input = ddata->input;

int i;



if (device\_may\_wakeup (dev)) {

for (i = 0; i < ddata->pdata->nbuttons; i++) {

struct gpio\_button\_data \*bdata = &ddata->data [i];

if (bdata->button->wakeup)

enable irq\_wake (bdata->irq);

\_

}

} else {

mutex lock (&input->mutex);

\_

if (input->users)

gpio\_keys\_close (input);

mutex unlock (&input->mutex);

\_

}

return 0;

}

static int gpio\_keys\_resume (struct device \*dev)

{

struct gpio\_keys\_drvdata \*ddata = dev\_get\_drvdata (dev);

struct input\_dev \*input = ddata->input;

int error = 0;

int i;



if (device\_may\_wakeup (dev)) {

for (i = 0; i < ddata->pdata->nbuttons; i++) {

struct gpio\_button\_data \*bdata = &ddata->data [i];

if (bdata->button->wakeup)

disable irq\_wake (bdata->irq);

\_

}

} else {

mutex lock (&input->mutex);

\_

if (input->users)

error = gpio\_keys\_open (input);

mutex unlock (&input->mutex);

\_

}

if (error)

return error;

gpio\_keys\_report\_state (ddata);

return 0;

}

#endif

static SIMPLE DEV PM OPS (gpio\_keys\_pm\_ops, gpio\_keys\_suspend,

\_ \_ \_

gpio\_keys\_resume);

static struct platform\_driver gpio\_keys\_device\_driver = {

.probe = gpio\_keys\_probe,

.remove = gpio\_keys\_remove,

.driver = {

.name = "gpio-keys",

.owner = THIS\_MODULE,

.of\_match\_table = of\_match\_ptr (gpio\_keys\_of\_match),

.pm = &gpio\_keys\_pm\_ops,

}

};

static int init gpio\_keys\_init (void)

\_\_

{

return platform\_driver\_register (&gpio\_keys\_device\_driver);

}

static void exit gpio\_keys\_exit (void)

\_\_

{

platform\_driver\_unregister (&gpio\_keys\_device\_driver);

}



late initcall (gpio\_keys\_init);

module\_exit (gpio\_keys\_exit);

MODULE LICENSE ("GPL");

MODULE\_AUTHOR ("Phil Blundell <pb@handhelds.org>");

MODULE DESCRIPTION ("Keyboard driver for GPIOs");

\_

MODULE ALIAS ("platform:gpio-keys");

设备树分析： gpio-keys {

compatible = "gpio-keys";

pinctrl-0 = <&pinctrl\_gpio\_keys>;

number1 {

label = "Number1 Button";

gpio-key,wakeup;

linux,code = <KEY\_1>; };

number2 {

label = "Number2 Button";

gpio-key,wakeup;

linux,code = <KEY\_2>; };

number3 {

label = "Number3 Button";

gpio-key,wakeup;

linux,code = <KEY\_3>; };

number4 {

label = "Number4 Button";

gpio-key,wakeup;

linux,code = <KEY\_4>; };

number5 {

label = "Number5 Button";

gpio-key,wakeup;

linux,code = <KEY\_5>; };

number6 {

label = "Number6 Button";

gpio-key,wakeup;

linux,code = <KEY\_6>; };

number7 {

label = "Number7 Button";

pinctrl-names = "default";

gpios = <&gpio1 21 1>;

gpios = <&gpio6 11 1>;

gpios = <&gpio1 20 1>;

gpios = <&gpio2 2 1>;

gpios = <&gpio2 6 1>;

gpios = <&gpio2 7 1>;

\_

gpios = <&gpio1 4 1>;

gpio-key,wakeup;

linux,code = <KEY\_7>;

};



number8 {

label = "Number8 Button";

gpios = <&gpio6 14 1>;

gpio-key,wakeup;

linux,code = <KEY\_8>;

};



number9 {

label = "Number9 Button";

gpios = <&gpio1 30 1>;

gpio-key,wakeup;

linux,code = <KEY\_9>;

};



escbutton {

label = "esc Button";

gpios = <&gpio2 16 1>;

gpio-key,wakeup;

linux,code = <KEY\_ESC>;

};



volume-up {

label = "Volume Up";

gpios = <&gpio2 5 1>;

gpio-key,wakeup;

linux,code = <KEY\_VOLUMEUP>;

};



volume-down {

label = "Volume Down";

gpios = <&gpio1 12 1>;

gpio-key,wakeup;

linux,code = <KEY\_VOLUMEDOWN>;

};

};

……



pinctrl\_gpio\_keys: gpio\_keysgrp { fsl,pins = <

MX6QDL\_PAD\_NANDF\_D7 GPIO2\_IO07 0x80000000

MX6QDL\_PAD\_NANDF\_D2 GPIO2\_IO02 0x80000000

MX6QDL\_PAD\_SD1\_DAT3 GPIO1\_IO21 0x80000000

MX6QDL\_PAD\_NANDF\_D6 GPIO2\_IO06 0x80000000

MX6QDL\_PAD\_NANDF\_CS0 GPIO6\_IO11 0x80000000

MX6QDL\_PAD\_SD1\_CLK GPIO1\_IO20 0x80000000

MX6QDL\_PAD\_GPIO\_4 GPIO1\_IO04 0x80000000

MX6QDL\_PAD\_NANDF\_CS1 GPIO6\_IO14 0x80000000

MX6QDL\_PAD\_ENET\_TXD0 GPIO1\_IO30 0x80000000

MX6QDL\_PAD\_EIM\_A22 GPIO2\_IO16 0x80000000

MX6QDL\_PAD\_NANDF\_D5 GPIO2\_IO05 0x80000000

MX6QDL\_PAD\_SD2\_DAT3 GPIO1\_IO12 0x80000000

>;

};

TEL：18500370067

第 169 页

MX6QDL\_PAD\_NANDF\_D7 GPIO2\_IO07 就是把 NANDF\_CLE 配置成 GPIO2\_IO7 功能，引脚复用。 NANDF\_D7 其他功能的定义可以查看 arch/arm/boot/dts/IMX6DL-pinfunc.h ，里面是 IMX6DL 引脚的所有配 置功能。例如：MX7QDL\_PAD\_NANDF\_D7\_\_NAND\_DATA07 就把引脚配置成 NAND\_DATA07 功能。#define MX6QDL\_PAD\_NANDF\_D7 SD2\_DATA7 就把引脚配置成 SD2\_DATA7。

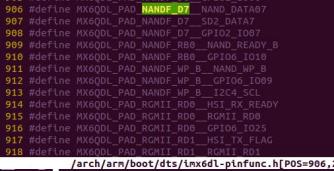


图 3.3.3 GPIO 宏定义

MX6QDL\_PAD\_NANDF\_CLE GPIO6\_IO07 0x80000000 LED1

MX6QDL\_PAD\_NANDF\_CS2 GPIO6\_IO15 0x80000000 LED2

MX6QDL\_PAD\_NANDF\_ALE GPIO6\_IO08 0x80000000 LED3

MX6QDL\_PAD\_NANDF\_RB0 GPIO6\_IO10 0x80000000 LED4

◆ 应用程序分析：

#include <stdio .h>

#include <sys/types .h>

#include <sys/stat.h>

#include <fcntl .h>

#include <linux/input .h>

#define NOKEY 0

int main (int argc,char \*argv [])

{

int keys\_fd;

char ret [2];

struct input\_event t;

keys\_fd = open (argv [1], O\_RDONLY);

if (keys\_fd<=0)

{

printf ("open %s device error !\n",argv [1]); return 0;

}

while (1) {

if (read(keys\_fd,&t,sizeof (t))==sizeof (t)) {

if (t .type==EV\_KEY)

if (t .value==0 || t.value==1)

printf ("key %d %s\n",t.code, (t.value)?"Pressed" :"Released");

} }

close (keys\_fd);



return 0;

}

5. 实验步骤

**5.1 实验目录**

/home/uptech/fsl-6dl-source/kernel-4.9.88/ /IMX6/SRC/exp/03\_keys/

**5.2 编译** **KEY 驱动程序**

1、进入实验目录：

root@uptech-virtual-machine:/home/uptech#**cd fsl-6dl-source/kernel-4.9.88**

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#**export ARCH=arm**

[root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#)**[make](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#) menuconfig**

2、配置内核

运行 make menuconfig(注意：如运行失败请加权限)之后，弹出如下的对话框，选择 Device Drivers ，按 回车进入。

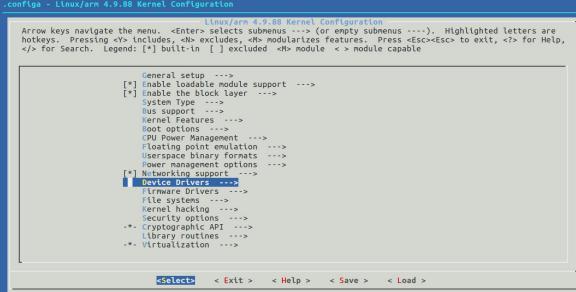


图 3.3.4 选择 Device Drivers 然后选择 Input device support 选项，按回车进入，如图 3.3.5 所示：



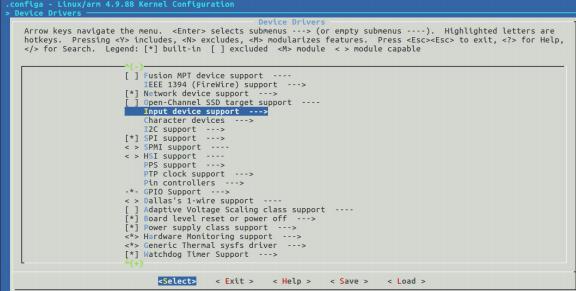


图 3.3.5 选择 Input device support 选择 Keyboards 选项，按回车进入，如图 3.3.6 所示：

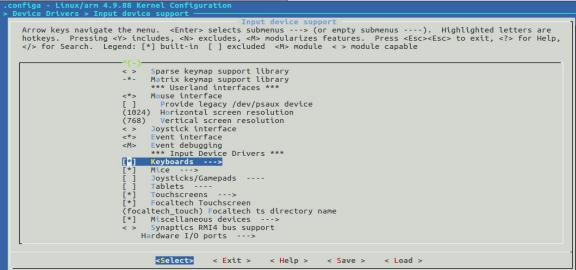


图 3.3.6 选择 Keyboards

选中 GPIO Buttons ，按键盘的空白键，将 GPIO Button 前面的可选框由‘\* ’切换为‘M ’，以模块加 载的方式进行配置，如图所示：

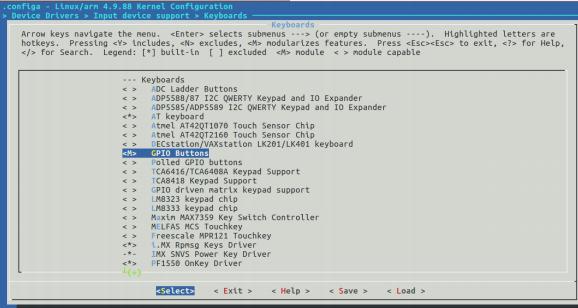


图 3.3.7 选择 GPIO Buttons

3、编译项目

[root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#) **make zImage**

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#

**make imx6dl-sabresd.dtb**

编译成功后会在源码目录的 arm/arch/boot/目录下生成内核压缩文件 zImage 、在 arch/arm/boot 下的 imx6dl-sabresd.dtb 文件,生成的 gpio-keys.ko 驱动在 fsl-6dl-source/kernel-4.9.88/drivers/input/keyboard 目录。 将编译生成的 zImage 、IMX6DL-sabresd.dtb 文件拷贝至出厂演示程序对应的目录中，重新烧写系统。

将编译生成的 gpio\_keys.ko 驱动文件拷贝至对应的实验文件夹中。

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88/drivers

/input/keyboard# **cp gpio\_keys.ko /IMX6/SRC/exp/driver/03\_keys/**

**5.3 编译** **KEY 应用测试程序**

1 、进入实验目录：

root@uptech-virtual-machine:~# cd IMX6/SRC/exp/driver/03\_keys

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys/# ls

Makefile key.c

2 、清除中间代码，重新编译

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# source

/opt/fsl-imx-wayland/4.9.88-2.0.0/environment-setup-cortexa9hf-neon-poky-linux-gnueabi

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# make clean

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# make

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# ls

key.c key.o Makefile key

当前目录下生成可执行程序 key。

**5.4 NFS 挂载实验目录测试**

1 、 启动 IMX6 实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

root@IMX6DLsabresd:~# mount -t nfs 192.168.12.157:/IMX6 /mnt/

2 、 进入串口终端的 NFS 共享实验目录。

root@IMX6DLsabresd:~# cd /mnt/SRC/exp/driver/03\_keys

root@IMX6DLsabresd:/mnt/SRC/exp/driver/03\_keys#insmod gpio\_keys.ko

input: gpio-keys.21 as /devices/soc0/gpio-keys.21/input/input6

可以通过 proc/bus/input/devices 查看输入设备信息：

root@IMX6DLsabresd:/mnt/SRC/exp/driver/03\_keys# **cat /proc/bus/input/devices**

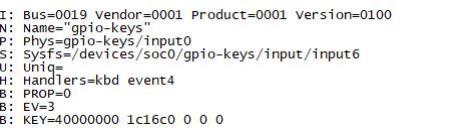


图 3.3.8 查询节点

从上面的打印信息,Name= ”gpio-keys ”可知，/dev/input/event4 是按键输入设备。

3 、 执行应用程序测试该驱动及设备

root@IMX6DLsabresd:/mnt/SRC/exp/driver/03\_keys# **./key /dev/input/event4** 执行应用程序后，按下 K7 按键可看到如下打印信息：



图 3.3.9 实验现象

105 代表是 KEY\_7 按键，可以在内核源码 arch/arm/boot/dts/include/dt-bindings/input 的 input.h 文件中查 看到 KEY\_7 的宏定义。Pressed 表示按键被按下，Released 表示按键被松开。

当然我们还可以使用内核自带的键盘程序，我们只需配置内核添加键盘驱动, 并在板文件添加相应的配 置代码，系统就可以读取按键值了。这样我们就不需要专门编写按键的驱动代码了。

CTRL+C 可以终止程序。



struct gpio\_button\_data \*bdata = &ddata->data [i];

bu f [ret++] = '\n ';

\* gpio\_keys\_enable\_button () - enables given GPIO button

\* @bdata: button data for button to be disabled \*

\* Enables given button pointed by @bdata. \*

\* Make sure that @bdata->disable\_lock is locked when entering

\* this function to avoid races with concurrent threads trying

\* to enable the same button at the same time.

\*/

static void gpio\_keys\_enable\_button (struct gpio\_button\_data \*bdata)

{

if (bdata->disabled) {

enable irq(bdata->irq);

bdata- = false;

}

}

/\*\*

\* gpio\_keys\_attr\_show\_helper () - fill in stringified bitmap of buttons

\* @ddata: pointer to drvdata

\* @bu f: buffer where stringified bitmap is written

\* @type : button type (%EV\_KEY, %EV\_SW)

\* @only\_disabled: does caller want only those buttons that are

\* currently disabled or all buttons that can be

\* disabled \*

\* This function writes buttons that can be disabled to @bu f. If

\* @only\_disabled is true, then @buf contains only those buttons

\* that are currently disabled . Returns 0 on success or negative

\* errno on failure.

\*/

static ssize\_t gpio\_keys\_attr\_show\_helper (struct gpio\_keys\_drvdata \*ddata,

char \*buf, unsigned int type,

bool only\_disabled)

{

int n\_events = get n events\_by\_type (type);

unsigned long \*bits;

ssize t ret;

int i;

bits = kcalloc (BITS\_TO\_LONGS (n\_events), sizeof (\*bits), GFP\_KERNEL);

if ( !bits)

return -ENOMEM;

for (i = 0; i < ddata->pdata->nbuttons; i++) {

if (bdata->button->type != type) continue;

if (only\_disabled && !bdata->disabled)

continue;

set\_bit (bdata->button->code, bits);

}

ret = bitmap\_scnlistprintf (buf, PAGE\_SIZE - 2, bits, n\_events);

bu f [ret] = '\0 '; kfree (bits);

return ret;

}

/\*\*

\* gpio\_keys\_attr\_store\_helper () - enable/disable buttons based on given bitmap

\* @ddata: pointer to drvdata

\* @bu f: buffer from userspace that contains stringified bitmap

\* @type : button type (%EV\_KEY, %EV\_SW)

\*

\* This function parses stringified bitmap from @buf and disables/enables

\* GPIO buttons accordingly. Returns 0 on success and negative error

\* on failure.

\*/

static ssize t gpio\_keys\_attr\_store\_helper (struct gpio\_keys\_drvdata \*ddata,

const char \*bu f, unsigned int type) {

int n\_events = get n events\_by\_type (type);

unsigned long \*bits;

ssize t error;

int i;

bits = kcalloc (BITS\_TO\_LONGS (n\_events), sizeof (\*bits), GFP\_KERNEL);

if ( !bits)

return -ENOMEM;

error = bitmap\_parselist (buf, bits, n\_events);

if (error)

goto out;

/\* First validate \*/

for (i = 0; i < ddata->pdata->nbuttons; i++) {

if (bdata->button->type != type) continue;

if (test\_bit (bdata->button->code, bits) &&

!bdata->button->can disable) { error = -EINVAL; \_

goto out;

}

}

mutex lock (&ddata->disable\_lock);

for (i = 0; i < ddata->pdata->nbuttons; i++) {

if (bdata->button->type != type) continue;

if (test\_bit (bdata->button->code, bits))

gpio\_keys\_disable\_button (bdata);

else

struct gpio\_button\_data \*bdata = &ddata->data [i];

struct gpio\_button\_data \*bdata = &ddata->data [i];

\_

\_

gpio\_keys\_enable\_button (bdata);

}

mutex\_unlock (&ddata->disable\_lock);

out:

kfree (bits);

return error;

}

#define ATTR\_SHOW\_FN (name, type, only\_disabled) \

static ssize\_t gpio\_keys\_show\_##name (struct device \*dev, \

struct device\_attribute \*attr, \

char \*buf) \

{ \

struct platform\_device \*pdev = to\_platform\_device (dev); \

struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata (pdev); \

\

return gpio\_keys\_attr\_show\_helper (ddata, bu f, \

type, only\_disabled); \

}

ATTR SHOW FN (keys, EV\_KEY, false);

\_ \_

ATTR\_SHOW\_FN (switches, EV\_SW, false);

ATTR\_SHOW\_FN (disabled\_keys, EV\_KEY, true);

ATTR\_SHOW\_FN (disabled\_switches, EV\_SW, true);

/\*

\* ATTRIBUTES: \*

\* /sys/devices/platform/gpio-keys/keys [ro]

\* /sys/devices/platform/gpio-keys/switches [ro]

\*/

static DEVICE\_ATTR(keys, S\_IRUGO, gpio\_keys\_show\_keys, NULL);

static DEVICE ATTR(switches, S\_IRUGO, gpio\_keys\_show\_switches, NULL);

#define ATTR\_STORE\_FN (name, type) \

static ssize\_t gpio\_keys\_store\_##name (struct device \*dev, \

struct device\_attribute \*attr, \

const char \*buf, \

size\_t count) \

{ \

struct platform\_device \*pdev = to\_platform\_device (dev); \

struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata (pdev); \

ssize t error; \

\

error = gpio\_keys\_attr\_store\_helper (ddata, bu f, type); \

if (error) \

return error; \

\

return count; \

}

ATTR\_STORE\_FN (disabled\_keys, EV\_KEY);

ATTR\_STORE\_FN (disabled\_switches, EV\_SW);

/\*

\* ATTRIBUTES:

\*

\* /sys/devices/platform/gpio-keys/disabled\_keys [rw]

\* /sys/devices/platform/gpio-keys/disables\_switches [rw]

\*/

static DEVICE ATTR(disabled\_keys, S\_IWUSR | S\_IRUGO,

\_

gpio\_keys\_show\_disabled\_keys,

gpio\_keys\_store\_disabled\_keys);

static DEVICE ATTR(disabled\_switches, S\_IWUSR | S\_IRUGO,

\_

gpio\_keys\_show\_disabled\_switches,

gpio\_keys\_store\_disabled\_switches);

static struct attribute \*gpio\_keys\_attrs [] = {

&dev attr keys .attr,

\_ \_

&dev attr switches .attr,

\_ \_

&dev attr disabled keys.attr,

\_ \_ \_

&dev attr disabled switches.attr,

\_ \_ \_

NULL,

};

static struct attribute\_group gpio\_keys\_attr\_group = {

.attrs = gpio\_keys\_attrs,

};

static void gpio\_keys\_gpio\_report\_event (struct gpio\_button\_data \*bdata)

{

const struct gpio\_keys\_button \*button = bdata->button;

struct input\_dev \*input = bdata->input;

unsigned int type = button->type ?: EV\_KEY; ^

int state = (gpio\_get\_value\_cansleep (button->gpio) ? 1 : 0)

button->active low;

\_

if (type == EV\_ABS) {

if (state)

input\_event (input, type, button->code, button->value);

} else {

input\_event (input, type, button->code, ! !state);

}

input\_sync (input);

}

static void gpio\_keys\_gpio\_work\_func (struct work\_struct \*work)

{

struct gpio\_button\_data \*bdata =

container of (work, struct gpio\_button\_data, work);

\_

gpio\_keys\_gpio\_report\_event (bdata);

if (bdata->button->wakeup)

pm\_relax (bdata->input->dev .parent);

}

static void gpio\_keys\_gpio\_timer (unsigned long \_data)

{

struct gpio\_button\_data \*bdata = (struct gpio\_button\_data \*)\_data;

schedule work (&bdata->work);

\_

}

static irqreturn\_t gpio\_keys\_gpio\_is r (int irq, void \*dev\_id) {

struct gpio\_button\_data \*bdata = dev\_id;

BUG\_ON (irq != bdata->irq);

if (bdata->button->wakeup)

pm\_stay\_awake (bdata->input->dev.parent);

if (bdata->timer\_debounce)

mod\_timer (&bdata->timer,

jiffies + msecs\_to\_jiffies (bdata->timer\_debounce));

else

schedule work (&bdata->work);

\_

return IRQ HANDLED;

}

static void gpio\_keys\_irq\_timer (unsigned long \_data) {

struct gpio\_button\_data \*bdata = (struct gpio\_button\_data \*)\_data;

struct input\_dev \*input = bdata->input;

unsigned long flags;

spin\_lock\_irqsave (&bdata->lock, flags);

if (bdata->key\_pressed) {

input\_event (input, EV\_KEY, bdata->button->code, 0);

input\_sync (input);

bdata->key\_pressed = false;

}

spin\_unlock\_irqrestore (&bdata->lock, flags);

}

static irqreturn\_t gpio\_keys\_irq\_is r (int irq, void \*dev\_id) {

struct gpio\_button\_data \*bdata = dev\_id;

const struct gpio\_keys\_button \*button = bdata->button;

struct input\_dev \*input = bdata->input;

unsigned long flags;

BUG\_ON (irq != bdata->irq);

spin\_lock\_irqsave (&bdata->lock, flags);

if ( !bdata->key\_pressed) {

if (bdata->button->wakeup)

pm\_wakeup\_event (bdata->input->dev.parent, 0);

input\_event (input, EV\_KEY, button->code, 1);

input\_sync (input);

if ( !bdata->timer\_debounce) {

input\_event (input, EV\_KEY, button->code, 0);

input\_sync (input);

goto out;

}

bdata->key\_pressed = true; }

if (bdata->timer\_debounce)

mod timer (&bdata->timer,

\_

jiffies + msecs\_to\_jiffies (bdata->timer\_debounce));

out:

spin\_unlock\_irqrestore (&bdata->lock, flags);

return IRQ HANDLED;

}

static int gpio\_keys\_setup\_key (struct platform\_device \*pdev,

struct input\_dev \*input,

struct gpio\_button\_data \*bdata,

const struct gpio\_keys\_button \*button)

{

const char \*desc = button->desc ? button->desc : "gpio\_keys";

struct device \*dev = &pdev->dev;

irq\_handler\_t isr;

unsigned long irqflags;

int irq, error;

bdata->input = input;

bdata->button = button;

spin\_lock\_init (&bdata->lock);

if (gpio\_is\_valid(button->gpio)) {

error = gpio\_request\_one (button->gpio, GPIOF\_IN, desc);

if (error < 0) {

dev err (dev, "Failed to request GPIO %d, error %d\n",

\_

button->gpio, error);

return error;

}

if (button->debounce\_interval) {

error = gpio\_set\_debounce (button->gpio,

button->debounce interval \* 1000);

/\* use timer if gpiolib doesn 't provide debounce \*/

if (error < 0)

bdata->timer\_debounce =

button->debounce interval;

}

irq = gpio\_to\_irq(button->gpio);

if (irq < 0) {

error = irq;

dev err (dev,

\_

"Unable to get irq number for GPIO %d, error %d\n",

button->gpio, error);

goto fail;

}

bdata->irq = irq;

INIT WORK (&bdata->work, gpio\_keys\_gpio\_work\_func);

\_

setup\_timer (&bdata->timer,

gpio\_keys\_gpio\_timer, (unsigned long)bdata);

is r = gpio\_keys\_gpio\_is r;

irqflags = IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING;

if (bdata->button->wakeup)

irqflags |= IRQF\_NO\_SUSPEND;

} else {

if ( !button->irq) {

dev\_err (dev, "No IRQ specified\n");

return -EINVAL; }

bdata->irq = button->irq;

if (button->type && button->type != EV\_KEY) {

dev\_err (dev, "Only EV\_KEY allowed for IRQ buttons.\n");

return -EINVAL;

}

bdata->timer\_debounce = button->debounce\_interval;

setup\_timer (&bdata->timer,

gpio\_keys\_irq\_timer, (unsigned long)bdata);

is r = gpio\_keys\_irq\_isr;

irqflags = 0;

}

input\_set\_capability (input, button->type ?: EV\_KEY, button->code);

/\*

\* If platform has specified that the button can be disabled,

\* we don 't want it to share the interrupt line.

\*/

if ( !button->can\_disable)

irqflags |= IRQF\_SHARED;

error = request\_any\_context\_irq(bdata->irq, is r, irqflags, desc, bdata); if (error < 0) {

dev err (dev, "Unable to claim irq %d; error %d\n",

bdata->irq, error);

goto fail;

}

return 0;

fail :

if (gpio\_is\_valid(button->gpio))

gpio\_free (button->gpio);

return error;

}

static void gpio\_keys\_report\_state (struct gpio\_keys\_drvdata \*ddata) {

struct input\_dev \*input = ddata->input; int i;

for (i = 0; i < ddata->pdata->nbuttons; i++) {

struct gpio\_button\_data \*bdata = &ddata->data [i];

if (gpio\_is\_valid(bdata->button->gpio))

gpio\_keys\_gpio\_report\_event (bdata);

}

input\_sync (input);

}

static int gpio\_keys\_open (struct input\_dev \*input)

{

struct gpio\_keys\_drvdata \*ddata = input\_get\_drvdata (input);

const struct gpio\_keys\_platform\_data \*pdata = ddata->pdata;

int error;

if (pdata->enable) {

error = pdata->enable (input->dev .parent);

if (error)

return error;

}

/\* Report current state of buttons that are connected to GPIOs \*/

gpio\_keys\_report\_state (ddata);

return 0;

}

static void gpio\_keys\_close (struct input\_dev \*input)

{

struct gpio\_keys\_drvdata \*ddata = input\_get\_drvdata (input);

const struct gpio\_keys\_platform\_data \*pdata = ddata->pdata;

if (pdata->disable)

pdata->disable (input->dev.parent);

}

/\*

\* Handlers for alternative sources of platform\_data

\*/

#ifdef CONFIG OF

/\*

\* Translate OpenFirmware node properties into platform\_data

\*/

static struct gpio\_keys\_platform\_data \*

gpio\_keys\_get\_devtree\_pdata (struct device \*dev)

{

struct device node \*node, \*pp;

\_

struct gpio\_keys\_platform\_data \*pdata;

struct gpio\_keys\_button \*button;

int error;

int nbuttons;

int i;

node = dev->of\_node;

if ( !node) {

error = -ENODEV;

goto err\_out;

}

nbuttons = of\_get\_child\_count (node);

if (nbuttons == 0) {

goto err\_out;

error = -ENODEV;

}

pdata = kzalloc (sizeof (\*pdata) + nbuttons \* (sizeof \*button),

GFP KERNEL);

\_

if ( !pdata) {

error = -ENOMEM;

goto err\_out;

}

pdata->buttons = (struct gpio\_keys\_button \*) (pdata + 1);

pdata->nbuttons = nbuttons;

pdata->rep = ! !of\_get\_property (node, "autorepeat", NULL);

i = 0;

for each child of node (node, pp) {

\_ \_ \_ \_

int gpio;

enum of gpio\_flags flags;

\_

if ( !of\_find\_property (pp, "gpios", NULL)) {

pdata->nbuttons--;

dev warn (dev, "Found button without gpios\n");

\_

continue;

}

gpio = of\_get\_gpio\_flags (pp, 0, &flags);

if (gpio < 0) {

error = gpio;

if (error != -EPROBE\_DEFER)

dev err (dev,

\_

"Failed to get gpio flags, error: %d\n",

error);

goto err\_free\_pdata;

}

button = &pdata->buttons [i++];

button->gpio = gpio;

button->active\_low = flags & OF\_GPIO\_ACTIVE\_LOW;

if (of\_property\_read\_u32 (pp, "linux,code", &button->code)) {

dev err (dev, "Button without keycode: 0x%x\n",

\_

button->gpio); error = -EINVAL;

goto err\_free\_pdata;

}

button->desc = of\_get\_property (pp, "label", NULL);

if (of\_property\_read\_u32 (pp, "linux,input-type", &button->type))

button->type = EV\_KEY;

button->wakeup = ! !of\_get\_property (pp, "gpio-key,wakeup", NULL);

if (of\_property\_read\_u32 (pp, "debounce-interval",

&button->debounce interval)) button->debounce\_interva = 5;

}



error = -EINVAL;

if (pdata->nbuttons == 0) {

goto err\_free\_pdata;

}

return pdata;

err free pdata :

\_ \_

kfree (pdata);

err out:

return ERR PTR (error);

\_

}

static struct of\_device\_id gpio\_keys\_of\_match [] = {

{ .compatible = "gpio-keys", },

{ },

};

MODULE DEVICE TABLE (of, gpio\_keys\_of\_match);

\_ \_

#else

static inline struct gpio\_keys\_platform\_data \*

gpio\_keys\_get\_devtree\_pdata (struct device \*dev)

{

return ERR PTR (-ENODEV);

}

#endif

static void gpio\_remove\_key (struct gpio\_button\_data \*bdata)

{

free irq(bdata->irq, bdata);

\_

if (bdata->timer\_debounce)

del timer sync (&bdata->timer);

\_ \_

cancel work sync (&bdata->work);

\_ \_

if (gpio\_is\_valid(bdata->button->gpio))

gpio\_free (bdata->button->gpio);

}

static int gpio\_keys\_probe (struct platform\_device \*pdev)

{

struct device \*dev = &pdev->dev;

const struct gpio\_keys\_platform\_data \*pdata = dev\_get\_platdata (dev);

struct gpio\_keys\_drvdata \*ddata;

struct input\_dev \*input;

int i, error;

int wakeup = 0;

if ( !pdata) {

pdata = gpio\_keys\_get\_devtree\_pdata (dev);

if (IS\_ERR(pdata))

return PTR ERR(pdata);

\_

}

ddata = kzalloc (sizeof (struct gpio\_keys\_drvdata) +

pdata->nbuttons \* sizeof (struct gpio\_button\_data),

GFP KERNEL);

\_

input = input\_allocate\_device ();

if ( !ddata || !input) {

dev err (dev, "failed to allocate state\n");

errr = -ENOMEM;

goto fail1;

}

ddata->pdata = pdata;

ddata->input = input;

mutex init (&ddata->disable\_lock);

\_

platform\_set\_drvdata (pdev, ddata);

input\_set\_drvdata (input, ddata);

/\* Enable auto repeat feature of Linux input subsystem \*/

input->name = pdata->name ? : pdev->name;

input->dev .parent = &pdev->dev;

input->close = gpio\_keys\_close;

input->id.bustype = BUS\_HOST;

input->id.product = 0x0001;

input->phys = "gpio-keys/input0";

input->open = gpio\_keys\_open;

input->id.version = 0x0100;

input->id.vendor = 0x0001;

if (pdata->rep)

set bit (EV\_REP, input->evbit);

\_\_ \_

for (i = 0; i < pdata->nbuttons; i++) {

const struct gpio\_keys\_button \*button = &pdata->buttons [i];

struct gpio\_button\_data \*bdata = &ddata->data [i];

error = gpio\_keys\_setup\_key (pdev, input, bdata, button);

if (error)

goto fail2;

if (button->wakeup) wakeup = 1;

}

error = sysfs\_create\_group (&pdev->dev.kobj, &gpio\_keys\_attr\_group);

if (error) {

dev err (dev, "Unable to export keys/switches, error: %d\n",

\_

error);

goto fail2;

}

error = input\_register\_device (input);

if (error) {

dev err (dev, "Unable to register input device, error: %d\n",

\_

error);

goto fail3;

}

device init wakeup (&pdev->dev, wakeup);

\_ \_



return 0;

fail3 :

sysfs\_remove\_group (&pdev->dev.kobj, &gpio\_keys\_attr\_group);

fail2 :

while (--i >= 0)

gpio\_remove\_key (&ddata->data [i]);

fail1 :

input\_free\_device (input);

kfree (ddata);

/\* If we have no platform data, we allocated pdata dynamically. \*/

if ( !dev\_get\_platdata (&pdev->dev))

kfree (pdata);

return error;

}

static int gpio\_keys\_remove (struct platform\_device \*pdev)

{

struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata (pdev);

struct input\_dev \*input = ddata->input;

int i;

sysfs\_remove\_group (&pdev->dev.kobj, &gpio\_keys\_attr\_group);

device init wakeup (&pdev->dev, 0);

\_ \_

for (i = 0; i < ddata->pdata->nbuttons; i++)

gpio\_remove\_key (&ddata->data [i]);

input\_unregister\_device (input);

/\* If we have no platform data, we allocated pdata dynamically. \*/

if ( !dev\_get\_platdata (&pdev->dev))

kfree (ddata->pdata);

kfree (ddata);

return 0;

}

#ifdef CONFIG PM SLEEP

static int gpio\_keys\_suspend (struct device \*dev)

{

struct gpio\_keys\_drvdata \*ddata = dev\_get\_drvdata (dev);

struct input\_dev \*input = ddata->input;

int i;



if (device\_may\_wakeup (dev)) {

for (i = 0; i < ddata->pdata->nbuttons; i++) {

struct gpio\_button\_data \*bdata = &ddata->data [i];

if (bdata->button->wakeup)

enable irq\_wake (bdata->irq);

\_

}

} else {

mutex lock (&input->mutex);

\_

if (input->users)

gpio\_keys\_close (input);

mutex unlock (&input->mutex);

\_

}

return 0;

}

static int gpio\_keys\_resume (struct device \*dev)

{

struct gpio\_keys\_drvdata \*ddata = dev\_get\_drvdata (dev);

struct input\_dev \*input = ddata->input;

int error = 0;

int i;



if (device\_may\_wakeup (dev)) {

for (i = 0; i < ddata->pdata->nbuttons; i++) {

struct gpio\_button\_data \*bdata = &ddata->data [i];

if (bdata->button->wakeup)

disable irq\_wake (bdata->irq);

\_

}

} else {

mutex lock (&input->mutex);

\_

if (input->users)

error = gpio\_keys\_open (input);

mutex unlock (&input->mutex);

\_

}

if (error)

return error;

gpio\_keys\_report\_state (ddata);

return 0;

}

#endif

static SIMPLE DEV PM OPS (gpio\_keys\_pm\_ops, gpio\_keys\_suspend,

\_ \_ \_

gpio\_keys\_resume);

static struct platform\_driver gpio\_keys\_device\_driver = {

.probe = gpio\_keys\_probe,

.remove = gpio\_keys\_remove,

.driver = {

.name = "gpio-keys",

.owner = THIS\_MODULE,

.of\_match\_table = of\_match\_ptr (gpio\_keys\_of\_match),

.pm = &gpio\_keys\_pm\_ops,

}

};

static int init gpio\_keys\_init (void)

\_\_

{

return platform\_driver\_register (&gpio\_keys\_device\_driver);

}

static void exit gpio\_keys\_exit (void)

\_\_

{

platform\_driver\_unregister (&gpio\_keys\_device\_driver);

}



late initcall (gpio\_keys\_init);

module\_exit (gpio\_keys\_exit);

MODULE LICENSE ("GPL");

MODULE\_AUTHOR ("Phil Blundell <pb@handhelds.org>");

MODULE DESCRIPTION ("Keyboard driver for GPIOs");

\_

MODULE ALIAS ("platform:gpio-keys");

设备树分析： gpio-keys {

compatible = "gpio-keys";

pinctrl-0 = <&pinctrl\_gpio\_keys>;

number1 {

label = "Number1 Button";

gpio-key,wakeup;

linux,code = <KEY\_1>; };

number2 {

label = "Number2 Button";

gpio-key,wakeup;

linux,code = <KEY\_2>; };

number3 {

label = "Number3 Button";

gpio-key,wakeup;

linux,code = <KEY\_3>; };

number4 {

label = "Number4 Button";

gpio-key,wakeup;

linux,code = <KEY\_4>; };

number5 {

label = "Number5 Button";

gpio-key,wakeup;

linux,code = <KEY\_5>; };

number6 {

label = "Number6 Button";

gpio-key,wakeup;

linux,code = <KEY\_6>; };

number7 {

label = "Number7 Button";

pinctrl-names = "default";

gpios = <&gpio1 21 1>;

gpios = <&gpio6 11 1>;

gpios = <&gpio1 20 1>;

gpios = <&gpio2 2 1>;

gpios = <&gpio2 6 1>;

gpios = <&gpio2 7 1>;

\_

gpios = <&gpio1 4 1>;

gpio-key,wakeup;

linux,code = <KEY\_7>;

};



number8 {

label = "Number8 Button";

gpios = <&gpio6 14 1>;

gpio-key,wakeup;

linux,code = <KEY\_8>;

};



number9 {

label = "Number9 Button";

gpios = <&gpio1 30 1>;

gpio-key,wakeup;

linux,code = <KEY\_9>;

};



escbutton {

label = "esc Button";

gpios = <&gpio2 16 1>;

gpio-key,wakeup;

linux,code = <KEY\_ESC>;

};



volume-up {

label = "Volume Up";

gpios = <&gpio2 5 1>;

gpio-key,wakeup;

linux,code = <KEY\_VOLUMEUP>;

};



volume-down {

label = "Volume Down";

gpios = <&gpio1 12 1>;

gpio-key,wakeup;

linux,code = <KEY\_VOLUMEDOWN>;

};

};

……



pinctrl\_gpio\_keys: gpio\_keysgrp { fsl,pins = <

MX6QDL\_PAD\_NANDF\_D7 GPIO2\_IO07 0x80000000

MX6QDL\_PAD\_NANDF\_D2 GPIO2\_IO02 0x80000000

MX6QDL\_PAD\_SD1\_DAT3 GPIO1\_IO21 0x80000000

MX6QDL\_PAD\_NANDF\_D6 GPIO2\_IO06 0x80000000

MX6QDL\_PAD\_NANDF\_CS0 GPIO6\_IO11 0x80000000

MX6QDL\_PAD\_SD1\_CLK GPIO1\_IO20 0x80000000

MX6QDL\_PAD\_GPIO\_4 GPIO1\_IO04 0x80000000

MX6QDL\_PAD\_NANDF\_CS1 GPIO6\_IO14 0x80000000

MX6QDL\_PAD\_ENET\_TXD0 GPIO1\_IO30 0x80000000

MX6QDL\_PAD\_EIM\_A22 GPIO2\_IO16 0x80000000

MX6QDL\_PAD\_NANDF\_D5 GPIO2\_IO05 0x80000000

MX6QDL\_PAD\_SD2\_DAT3 GPIO1\_IO12 0x80000000

>;

};

MX6QDL\_PAD\_NANDF\_D7 GPIO2\_IO07 就是把 NANDF\_CLE 配置成 GPIO2\_IO7 功能，引脚复用。 NANDF\_D7 其他功能的定义可以查看 arch/arm/boot/dts/IMX6DL-pinfunc.h ，里面是 IMX6DL 引脚的所有配 置功能。例如：MX7QDL\_PAD\_NANDF\_D7\_\_NAND\_DATA07 就把引脚配置成 NAND\_DATA07 功能。#define MX6QDL\_PAD\_NANDF\_D7 SD2\_DATA7 就把引脚配置成 SD2\_DATA7。

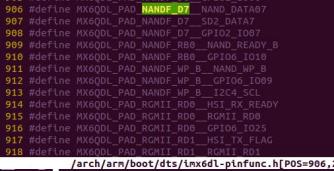


图 3.3.3 GPIO 宏定义

MX6QDL\_PAD\_NANDF\_CLE GPIO6\_IO07 0x80000000 LED1

MX6QDL\_PAD\_NANDF\_CS2 GPIO6\_IO15 0x80000000 LED2

MX6QDL\_PAD\_NANDF\_ALE GPIO6\_IO08 0x80000000 LED3

MX6QDL\_PAD\_NANDF\_RB0 GPIO6\_IO10 0x80000000 LED4

◆ 应用程序分析：

#include <stdio .h>

#include <sys/types .h>

#include <sys/stat.h>

#include <fcntl .h>

#include <linux/input .h>

#define NOKEY 0

int main (int argc,char \*argv [])

{

int keys\_fd;

char ret [2];

struct input\_event t;

keys\_fd = open (argv [1], O\_RDONLY);

if (keys\_fd<=0)

{

printf ("open %s device error !\n",argv [1]); return 0;

}

while (1) {

if (read(keys\_fd,&t,sizeof (t))==sizeof (t)) {

if (t .type==EV\_KEY)

if (t .value==0 || t.value==1)

printf ("key %d %s\n",t.code, (t.value)?"Pressed" :"Released");

} }

close (keys\_fd);



return 0;

}

5. 实验步骤

**5.1 实验目录**

/home/uptech/fsl-6dl-source/kernel-4.9.88/ /IMX6/SRC/exp/03\_keys/

**5.2 编译** **KEY 驱动程序**

1、进入实验目录：

root@uptech-virtual-machine:/home/uptech#**cd fsl-6dl-source/kernel-4.9.88**

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#**export ARCH=arm**

[root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#)**[make](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#) menuconfig**

2、配置内核

运行 make menuconfig(注意：如运行失败请加权限)之后，弹出如下的对话框，选择 Device Drivers ，按 回车进入。

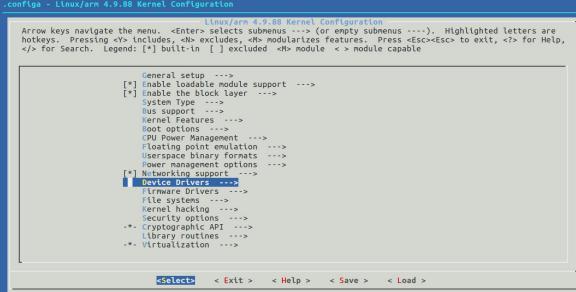


图 3.3.4 选择 Device Drivers 然后选择 Input device support 选项，按回车进入，如图 3.3.5 所示：



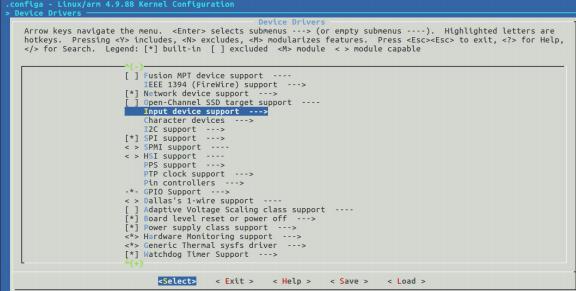


图 3.3.5 选择 Input device support 选择 Keyboards 选项，按回车进入，如图 3.3.6 所示：

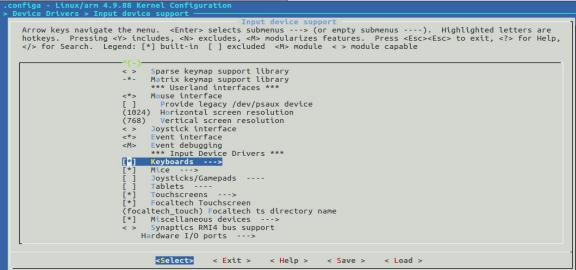


图 3.3.6 选择 Keyboards

选中 GPIO Buttons ，按键盘的空白键，将 GPIO Button 前面的可选框由‘\* ’切换为‘M ’，以模块加 载的方式进行配置，如图所示：

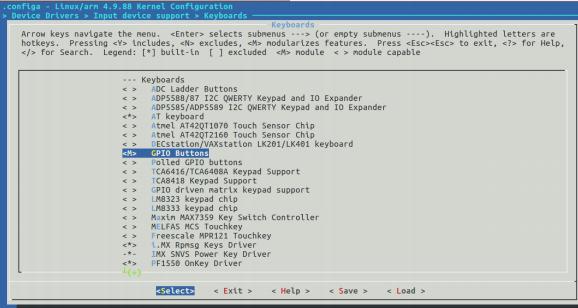


图 3.3.7 选择 GPIO Buttons

3、编译项目

[root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#](mailto:root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-3.14.28#) **make zImage**

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#

**make imx6dl-sabresd.dtb**

编译成功后会在源码目录的 arm/arch/boot/目录下生成内核压缩文件 zImage 、在 arch/arm/boot 下的 imx6dl-sabresd.dtb 文件,生成的 gpio-keys.ko 驱动在 fsl-6dl-source/kernel-4.9.88/drivers/input/keyboard 目录。 将编译生成的 zImage 、IMX6DL-sabresd.dtb 文件拷贝至出厂演示程序对应的目录中，重新烧写系统。

将编译生成的 gpio\_keys.ko 驱动文件拷贝至对应的实验文件夹中。

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88/drivers

/input/keyboard# **cp gpio\_keys.ko /IMX6/SRC/exp/driver/03\_keys/**

**5.3 编译** **KEY 应用测试程序**

1 、进入实验目录：

root@uptech-virtual-machine:~# cd IMX6/SRC/exp/driver/03\_keys

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys/# ls

Makefile key.c

2 、清除中间代码，重新编译

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# source

/opt/fsl-imx-wayland/4.9.88-2.0.0/environment-setup-cortexa9hf-neon-poky-linux-gnueabi

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# make clean

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# make

root@uptech-virtual-machine:/IMX6/SRC/exp/driver/03\_keys# ls

key.c key.o Makefile key

当前目录下生成可执行程序 key。

**5.4 NFS 挂载实验目录测试**

1 、 启动 IMX6 实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

root@IMX6DLsabresd:~# mount -t nfs 192.168.12.157:/IMX6 /mnt/

2 、 进入串口终端的 NFS 共享实验目录。

root@IMX6DLsabresd:~# cd /mnt/SRC/exp/driver/03\_keys

root@IMX6DLsabresd:/mnt/SRC/exp/driver/03\_keys#insmod gpio\_keys.ko

input: gpio-keys.21 as /devices/soc0/gpio-keys.21/input/input6

可以通过 proc/bus/input/devices 查看输入设备信息：

root@IMX6DLsabresd:/mnt/SRC/exp/driver/03\_keys# **cat /proc/bus/input/devices**

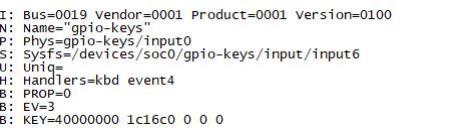


图 3.3.8 查询节点

从上面的打印信息,Name= ”gpio-keys ”可知，/dev/input/event4 是按键输入设备。

3 、 执行应用程序测试该驱动及设备

root@IMX6DLsabresd:/mnt/SRC/exp/driver/03\_keys# **./key /dev/input/event4** 执行应用程序后，按下 K7 按键可看到如下打印信息：



图 3.3.9 实验现象

105 代表是 KEY\_7 按键，可以在内核源码 arch/arm/boot/dts/include/dt-bindings/input 的 input.h 文件中查 看到 KEY\_7 的宏定义。Pressed 表示按键被按下，Released 表示按键被松开。

当然我们还可以使用内核自带的键盘程序，我们只需配置内核添加键盘驱动, 并在板文件添加相应的配 置代码，系统就可以读取按键值了。这样我们就不需要专门编写按键的驱动代码了。

CTRL+C 可以终止程序