全 志 科 技
Allwinner Technology

A20 平台 init-input 说明文档

V1.0

2013-06-17

Revision History

Version	Date	Changes compared to previous issue
V1.0	2013-06-17	Initial version

目录

1. 前言

1.1 编写目的

本文首先介绍了 init-input 模块的作用以及说明相关的接口，供驱动移植人员快速的使用。

（由于文档不断补充，代码也不断更新，有些地方可能和实际代码中有细微差别，请注意）

1.2 适用范围

适用于 A20 对应平台。

1.3 相关人员

项目中 input 设备驱动的开发，维护以及使用人员应认真阅读该文档。

1.4 文档介绍

本文主要针对 init-input 模块进行相关的介绍，对其中的关键数据结构以及所用接口进行清晰的描述，同时举例子说明相关接口的使用方法。

2. 模块介绍

2.1 模块功能介绍

init-input 模块（即输入设备初始化模块），该模块的主要功能为向其他输入设备提供以下统一功能。

Script 接口提供了解析 sys_config.fex 脚本的功能。

申请以及释放硬件 gpio 资源。

ctp 相关的 int(中断引脚），wakeup(复位引脚)硬件操作的接口封装。

2.2 相关术语介绍

Script 脚本：指的是打包到 img 中的 sys_config.fex 文件. 包含系统各模块配置参数.

Script 接口：指对 sys_config.fex 进行解析的函数.

2.3 模块源码介绍

init-input 模块的源码位于 input 目录下，如下所示：

3.3 内核：\lichee\linux-3.3\drivers\input\init-input.c

3.4 内核：\lichee\linux-3.3\drivers\input\init-input.c

包含该模块的头文件为：init-input.h，目录位置如下：

3.3 内核：\lichee\linux-3.3\include\linux\init-input.h

3.4 内核：\lichee\linux-3.4\include\linux\init-input.h

使用该文件提供的函数时，请包含头文件 init-input.h。

2.4 模块 menuconfig 配置

Init-input 为设备获取配置参数以及申请 gpio 资源的重要文件，建议编译进内核中，默认情况下该文件为编译近内核。

对于 init-input 的内核配置，可到 linux-3.x 目录下通过命令 make ARCH=arm menuconfig 进入配置主界面，以 linux3.4 为例子进行说明。并按以下步骤操作：

首先，选择 Device Drivers 选项进入下一级配置，如图 1 所示：

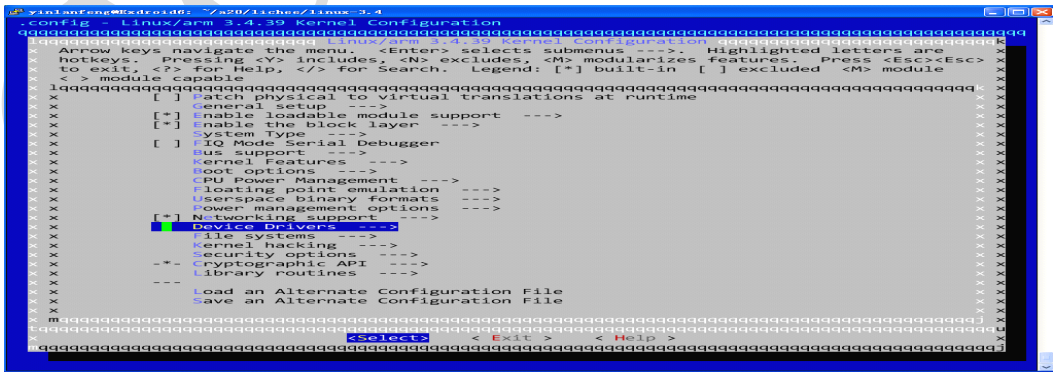

图 1 Device Drivers 选项配置

进入 Device Drivers 配置后，选择 Input device support 选项，如图 2 所示

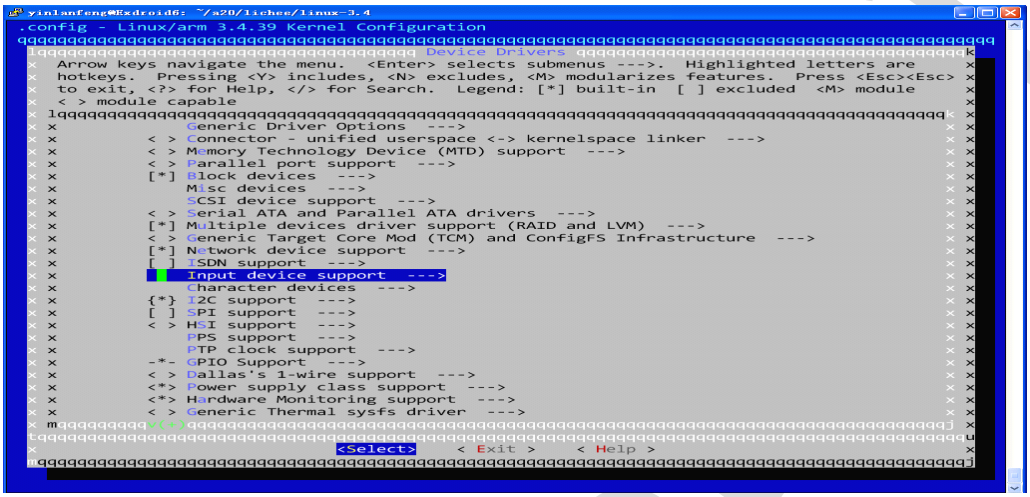

图 2 Input device support 选项配置

将 init device 选择编译进内核，如下所示：

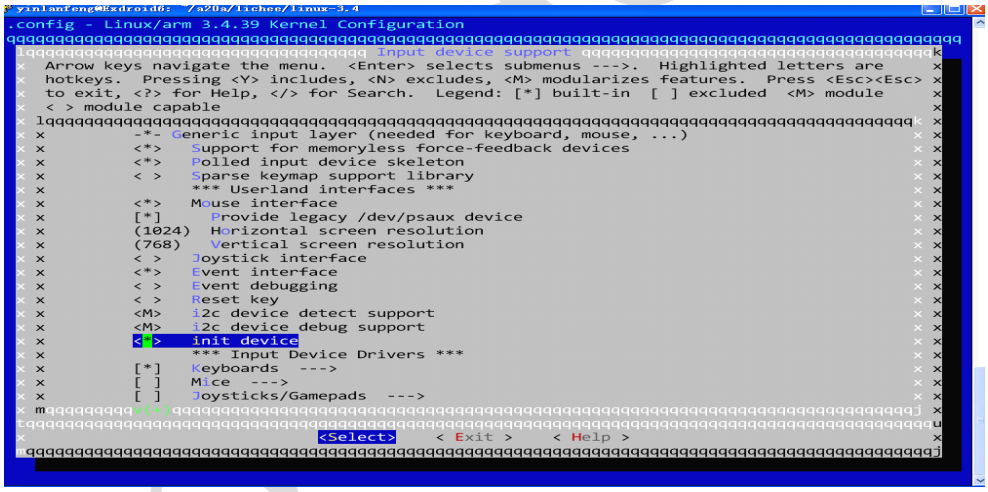

图 3 将 init-input 编译进内核

3. init-input 模块详细讲解

3.1 关键数据结构介绍

Init-input 模块中定义了一些数据结构用于存放获取到的 sysconfig.fex 参数，某些输入设备因为使用较少，有可能没有定义其相关的数据结构，使用时，请按照已使用的例子，自行添加相关数据结构。下面将一一介绍。

3.1.1 enum input_sensor_type

```
enum input_sensor_type{
        CTP_TYPE,
        GSENSOR_TYPE,
        GYR_TYPE,
        IR_TYPE,
        LS_TYPE,
        COMPASS_TYPE,
        MOTOR_TYPE
};
```

列举了一些输入设备类型，用于区分使用接口以及获取输入设备 sysconfig.fex 存放数据结构体信息，函数接口的选择等。

名称	作用
CTP_TYPE	ctp 设备类型
GSENSOR_TYPR	Gsensor 设备类型
GYR_TYPE	陀螺仪设备类型
IR_TYPE	IR 设备类型
LS_TYPE	光传感器设备类型
COMPASS_TYPE	Compass 设备类型
MOTOR_TYPE	振动器设备类型

3.1.2 struct ctp_config_info

该结构体用于存放 ctp 参数的数据结构名称。定义如下：

```
struct ctp_config_info{
        enum input_sensor_type    input_type;
        int         ctp_used;
```

```
        __u32     twi_id;
        int       screen_max_x;
        int       screen_max_y;
        int       revert_x_flag;
        int       revert_y_flag;
        int       exchange_x_y_flag;
        u32       int_number;
        u32       wakeup_number;
#ifdef TOUCH_KEY_LIGHT_SUPPORT
          u32       light_number;
#endif
};
```

名称	作用
input_type	Input 设备类型，ctp 的应为：CTP_TYPE ,主要用于在各接口获得整个结构体的信息
ctp_used	ctp_para 中，ctp_used 参数值，用于设备驱动中判断是否继续加载设备驱动
twi_id	ctp_para 中，ctp_twi_id 参数值，用于 detect 函数中
screen_max_x	ctp_para 中，ctp_screen_max_x 参数值,ctp 面板的 X 轴最大分辨率
screen_max_y	ctp_para 中，ctp_screen_max_y 参数值,ctp 面板的 Y 轴最大分辨率
revert_x_flag	ctp_para 中，ctp_revert_x_flag 参数值,ctp 面板的 X 轴方向，可配置值为 0，1。当前获取值为 1 时，发现 X 轴方向反置，在将该参数值替换为 0 时，即可以调整 X 轴方向
revert_y_flag	ctp_para 中，ctp_revert_y_flag 参数值,ctp 面板的 Y 轴方向，可配置值为 0，1。当前获取值为 1 时，发现 Y 轴方向反置，在将该参数值替换为 0，即可以调整 Y 轴方向
exchange_x_y_flag	ctp_para 中，ctp_exchange_x_y_flag 参数值,ctp 面板 X，Y 轴互换标志，可配置值为 0，1。当前获取值为 1 时，发现 X，Y 轴方向反了，在将该参数值替换为 0 时，即可以将 X，Y 轴的方向互换。
int_number	ctp_para 中，ctp_int_port 参数值,ctp 模块中断引脚号
wakeup_number	ctp_para 中，ctp_wakeup 参数值,ctp 模块复位引脚号
light_number	ctp_para 中，ctp_light 参数值，ctp 模块按键灯引脚号，当使用按键灯时使用，目前该参数均不使用。

3.1.3 sensor_config_info

```
struct sensor_config_info{
    enum input_sensor_type input_type;
    int sensor_used;
    __u32 twi_id;
};
```

该结构体为 sensors 设备类型，目前使用的 sensor 均使用轮询的方法获得数据。

名称	作用
input_type	Sensor 输入设备类型，根据驱动需要进行设置
sensor_use	Sysconfig.fex 中 xxx_para 的 xxx_used 参数值，表示该 sensor 是否启用
twi_id	Sysconfig.fex 中 xxx_para 的 xxx_twi_id 参数值，表示该 sensor 是使用的 i2c 组别号，在驱动的 detect 函数中使用。

3.1.4 struct ir_config_info

```
struct ir_config_info{
    enum input_sensor_type input_type;
    int ir_used;
    struct gpio_config ir_gpio;
    unsigned long rate;
```

该结构体用于存放 IR 模块的相关参数值。

名称	作用
input_type	输入设备类型，IR 应为 IR__TYPE
ir_used	Sysconfig.fex 中 ir_para 的 ir_used 参数值，表示该 IR 是否启用
ir_gpio	Sysconfig.fex 中 ir_para 的 ir_rx 参数值，rx 使用的 gpio 号
rate	目前尚未使用

3.2 script 接口介绍

3.2.1 int input_fetch_sysconfig_para

函数原型如下：

```
int input_fetch_sysconfig_para(enum input_sensor_type *input_type)
```

input_type：输入设备类型

返回值：成功返回 0，失败或者该设备的 xxx_used 不为 1 时，返回-1；

该函数使用输入的类型获取相关的数据结构体，实现了常使用的输入设备的具体函数，某些不经常使用的设备，可能没写出具体的实现函数，如果使用，请按照已经支持的设备写出需要的设备的具体实现。

目前已经实现的获取 sysconfig.fex 下配置参数的如下所示：

```
int (* const fetch_sysconfig_para[])(enum input_sensor_type *input_type) = {
        ctp_fetch_sysconfig_para,
        gsensor_fetch_sysconfig_para,
        gyr_fetch_sysconfig_para,
        ir_fetch_sysconfig_para,
        ls_fetch_sysconfig_para,
        e_compass_fetch_sysconfig_para
};
```

使用该函数时，注意 input_sensor_type 中设备的类型应与实际的实现函数对应，否则将可以读取错误。

3.2.2 script_item_u get_para_value

函数原型如下：

```
script_item_u get_para_value(char* keyname, char* subname)
```

根据主键值以及子健值对系统提供的函数：script_get_item 进行封装来获取配置参数，返回 script_item_u 数据结构数值。

keyname：主键值

Subname：子健值

返回值： 返回获取到的 script_item_u 结构体数据。

例子：需要获取 ctp_para 下 ctp_used 的数值。

则使用的语句如下：

```
.....
 int ctp_used;
script_item_u      val;
 val = get_para_value("ctp_para", "ctp_used");
  ctp_ used = val.val;
    .....
```

3.2.3 void get_str_para

函数原型如下：

```
void get_str_para(char* name[], char* value[], int num)
```

该函数可用于获取 sysconfig.fex 下 char 类型的参数值。

name：用于存放需要获取的主键以及子健名称。第一个必须放置主键名称。

value：依照 name 的顺序存取获取的 char 类型数据。

num：需要获取多少个 char 的内容，为子健的个数值。

例子 1：需要获取 ctp_para 下的 ctp_name 值，并将获取到的值放置在 cfgname 下。则使用语句如下所示：

```
.....
char *name[] = {
        "ctp_para",
        "ctp_name"
        };
 char * cfgname;
get_str_para(name, &cfgname,1);
.....
```

例子 2：若 ctp_para 下存在 ctp_name1,ctp_name2,ctp_name3,ctp_name4 的 char 类型数据，并将获取到的数据存在到 name1，name2，name3，name4 下。则使用的语句如下：

```
.....
char *getname[] = {
        "ctp_para",
        "ctp_name1",
        "ctp_name2",
        "ctp_name3",
        "ctp_name4"
        };
    char * value[4];
    char * name1, * name2, * name3,* name4;
get_str_para(getname, value,4);
name1 = value[0];
name2 = value[1];
name3 = value[2];
name4 = value[3];
.....
```

3.2.4 void get_int_para

函数的原型如下：

```
void get_int_para(char* name[], int value[], int num)
```

name：用于存放需要获取的主键以及子健名称。第一个必须放置主键名称。
value：依照 name 的顺序存取获取的 int 类型数据。
num：需要获取多少个 int 的内容，为子健的个数值。
例子 1：需要获取 ctp_para 下的 ctp_used 值，并将获取到的值放置在 used 下。则使用语句如下所示：

```
        ......

        char *name[] = {
                "ctp_para",
                "ctp_used"
                };
        int used;
        get_int_para(name, &used, 1);
        ......
```

例子 2：若需要获取 ctp_para 下存在 ctp_used, ctp_twi_id, ctp_screen_max_x, ctp_screen_max_y, ctp_revert_x_flag, ctp_revert_y_flag, ctp_exchange_x_y_flag 等 int 类型数据，并将获取到的数据存在到 ctp_used, twi_id, screen_max_x, screen_max_y, revert_x_flag,
revert_y_flag, exchange_x_y_flag 下。则使用的语句如下：

```
    ......
    int value[7] = {0};
     char *name[11] = {
                "ctp_para",
                "ctp_used",
                "ctp_twi_id",
                "ctp_screen_max_x",
                "ctp_screen_max_y",
                "ctp_revert_x_flag",
                "ctp_revert_y_flag",
                "ctp_exchange_x_y_flag",
    };
```

```
get_int_para(name, &value, 7);
 ctp_used                = value[0];
twi_id                   = value[1];
screen_max_x             = value[2];
screen_max_y             = value[3];
revert_x_flag            = value[4];
revert_y_flag            = value[5];
exchange_x_y_flag = value[6];
.....
```

3.2.5 void get_pio_para

函数原型如下：

```
void get_pio_para(char* name[], struct gpio_config value[], int num)
```

name：用于存放需要获取的主键以及子健名称。第一个必须放置主键名称。
value：依照 name 的顺序存取获取的 int 类型数据。
num：需要获取多少个 int 的内容，为子健的个数值。
例子 1：如需要获取 ctp_para 下的 ctp_int_port 的 gpio，则使用该语句为：

```
.....
char *name[] = {
                        "ctp_para",
                        "ctp_int_port"
                        };
            struct gpio_config int_gpio;
            int int_number;
            get_pio_para(name, &int_gpio, 1);
              int_number = int_gpio.gpio;
              ......
```

例子 2：如需要获取 ctp_para 下的 ctp_int_port ,ctp_wakeup 的 gpio，则使用该语句为：

```
.....
char *name[] = {
                        "ctp_para",
                        "ctp_int_port",
```

```
                    "ctp_wakeup"
                    };
            struct gpio_config gpio[2];
            int int_number, wakeup_number;
            get_pio_para(name, gpio, 2);
            int_number = gpio[0].gpio;
            wakeup_number = gpio[1].gpio;
    ......
```

3.3 i2c 通信相关接口

3.3.1 sw_i2c_write_bytes

函数原型如下：

```
int sw_i2c_write_bytes(struct i2c_client *client, uint8_t *data, uint16_t len)
```

client：存放地址的 i2c_client 结构体
data：存放需要写入的数据
len：数据的长度
返回值：成功返回 1，失败返回负数。
例子：寄存器地址写在数组的前面。

```
u8 data[2] = {0x80, 0x01, 0x55};
ret = i2c_write_bytes(client, ata, 3);
```

3.3.2 i2c_read_bytes_addr8

函数原型如下：

```
int i2c_read_bytes_addr8(struct i2c_client *client, uint8_t *buf, uint16_t len)
```

client：存放地址的 i2c_client 结构体
bug：存放读取到的据
len：数据的长度
注意：bug 的第一个数据（bug[0]）为需要读取的起始寄存器地址。读取回来的第一个数据（bug[0]）仍为寄存器地址，bug[1]开始为从 bug[0]寄存器开始的数据。
例子：读取 0x80 寄存器起始 32 个数据。

```
        ......
    u8 value[33] = {0x80};
```

```
        i2c_read_bytes_addr8(client, value, 33);
//数据的长度为需要读取的数据的个数与寄存器长度之和，即 32 + 1
        ......
```

3.3.3 i2c_read_bytes_addr16

函数原型如下：

```
int i2c_read_bytes_addr16(struct i2c_client *client, uint8_t *buf, uint16_t len)
```

client：存放地址的 i2c_client 结构体
bug：存放读取到的据
len：数据的长度
注意：bug 的第一与第二个数据（bug[0]，bug[1]）为需要读取的起始寄存器地址。读取回来的第一与第二数据（bug[0]，bug[1]）仍为寄存器地址，bug[2]开始为从 16 位寄存器(bug[0],bug[1])寄存器开始的数据。
例子：读取 0x715 寄存器起始 32 个数据。

```
        ......
        u8 value[34] = {0x07,0x15};
        i2c_read_bytes_addr8(client, value, 34);
//数据的长度为需要读取的数据的个数与寄存器长度之和，即 32 + 2
        ......
```

3.4 ctp 接口介绍

3.4.1 ctp_get_int_port_rate

函数原型如下：

```
bool ctp_get_int_port_rate(u32 gpio, u32 *clk)
```

功能：获取 CTP IRQ 频率
参数： CLK ：存放获取到的 clk 值
GPIO：INT 引脚号
返回值：true，读取成功。
false，读取失败。

3.4.2 ctp_set_int_port_rate

函数原型如下：

```
bool ctp_set_int_port_rate(u32 gpio, u32 clk)
```

功能：设置 CTP IRQ 频率

参数： CLK ：需要设置的 clk 值。clk= 0 时，clk 为 32Khz；clk= 1 时，clk 为 24Mhz。

GPIO：INT 引脚号

返回值：true，设置成功。

false，设置失败。

3.4.3 ctp_get_int_port_deb

函数原型如下：

```
bool ctp_get_int_port_deb(u32 gpio, u32 *clk_pre_scl)
```

功能：获取 CTP IRQ DEB 值(即 int 时钟的分频系数)

参数： CLK_PRE_SCL ：获取到的 DEB 值

GPIO：INT 引脚号

返回值：true，设置成功。

false，设置失败。

3.4.5 ctp_set_int_port_deb

函数原型如下：

```
bool ctp_set_int_port_deb(u32 gpio, u32 clk_pre_scl)
```

功能：设置 CTP IRQ DEB 值(即 int 时钟的分频系数，设置值 n 为 0 到 7，即将时钟分频到 2 的 n 次方)

参数： CLK_PRE_SCL ：设置的 DEB 值

GPIO：INT 引脚号

返回值：true，设置成功。

false，设置失败。

3.4.6 ctp_wakeup

函数原型如下所示：

```
int ctp_wakeup(u32 gpio, int status, int ms)
```

gpio: 为 wakeup 引脚的 gpio 编号

status：为输出的 gpio 的状态，0 表示低电平，1 表示高电平。

ms：延时时间。

Wakeup 的引脚号存在与 struct ctp_config_info 结构体中。当 ms 为 0 的时候，表示将 gpio 设置为 status 状态。当 ms 不为 0 时表示将 gpio 设置为 ms 后，在将

其设置为“-ms”。
例子：

```
ctp_wakeup(config_info.wakeup_number, 0, 10);//将 wakeup 引脚输出低 10ms 之后输出高。
ctp_wakeup(config_info.wakeup_number, 1, 20);//将 wakeup 引脚输出高 20ms 之后输出低。
ctp_wakeup(config_info.wakeup_number, 0, 0);//将 wakeup 引脚输出低。
ctp_wakeup(config_info.wakeup_number, 1, 0);//将 wakeup 引脚输出高。
```

3.5 申请与释放 gpio 接口介绍

3.5.1 input_init_platform_resource

函数原型如下：

```
int input_init_platform_resource(enum input_sensor_type *input_type)
```

input_type :输入设备类型
返回值：申请成功返回 0，失败返回-1；
该函数用于申请 gpio 资源，当没有 gpio 时，此函数设置为空。注意 input_sensor_type 中设备的类型应与实际的实现函数对应，否则将可以读取错误。
目前已经实现的获取 sysconfig.fex 下申请 gpio 资源的如下所示：

```
int (*init_platform_resource[])(enum input_sensor_type *input_type) = {
    ctp_init_platform_resource,
    gsensor_init_platform_resource,
    gyr_init_platform_resource,
    ir_init_platform_resource,
    ls_init_platform_resource,
    e_compass_init_platform_resource
};
```

3.5.2 input_free_platform_resource

函数原型如下：

```
void input_free_platform_resource(enum input_sensor_type *input_type)
```

input_type :输入设备类型

该函数用于释放申请 gpio 资源，当没有申请 gpio 资源时，此函数设置为空。注意 input_sensor_type 中设备的类型应与实际的实现函数对应，否则将可以读取错误。

目前已经实现的获取 sysconfig.fex 下释放 gpio 资源的如下所示：

```
void (*free_platform_resource[])(void) = {
    ctp_free_platform_resource,
    gsensor_free_platform_resource,
    gyr_free_platform_resource,
    ir_free_platform_resource,
    ls_free_platform_resource,
    e_compass_free_platform_resource
};
```

4. 使用示例

4.1 使用说明

输入设备驱动中使用 init-input 中的相关接口解析 sysconfig.fex 时，需要在设备驱动中添加头文件 init-input.h，即添加如下语句：

```
#include <linux/init-input.h>
```

4.2 ctp 使用示例说明

ctp 模块中，除了有参数的获取外，还有 gpio 的申请与释放。Gpio 的释放应放置在最后面进行。以 gt82x 驱动中为例子进行说明，如下所示：

```
.....
static struct ctp_config_info config_info = {
    .input_type = CTP_TYPE,
};//第一步：定义 ctp_config_info 结构体，并且将 input_type 赋值为 CTP_TYPE
.....
static int __devinit goodix_ts_init(void)
{
 .....
if (input_fetch_sysconfig_para(&(config_info.input_type))) {//第二步
        printk("%s: ctp_fetch_sysconfig_para err.\n", __func__);
        return 0;
    } else {
        ret = input_init_platform_resource(&(config_info.input_type));//第三步
        if (0 != ret) {
            printk("%s:ctp_ops.init_platform_resource err. \n", __func__);
        }
    }
    if(config_info.ctp_used == 0){//第四步
        printk("*** ctp_used set to 0 !\n");
        printk("*** if use ctp,please put the sys_config.fex ctp_used set to 1. \n");
        return 0;
```

```
    }
        ......
}
static void __exit goodix_ts_exit(void)
{
    printk("==goodix_ts_exit==\n");
    i2c_del_driver(&goodix_ts_driver);
    input_free_platform_resource(&(config_info.input_type));   //第五步
    return;
}
```

步骤如下：

第一步：定义 ctp_config_info 结构体，并且将 input_type 赋值为 CTP_TYPE

第二步：init 函数中调用 input_fetch_sysconfig_para 函数获取 sysconfig.fex 下的配置参数。

第三步：若能正确的获取参数，调用 input_init_platform_resource 接口申请 gpio 资源。

第四步：判断获取到的 ctp_used 是否为 1，若不为 1，则直接返回。

第五步：当该驱动退出时，应该在 exit 函数中调用 input_free_platform_resource 函数将申请的 gpio 资源释放掉。

4.3 gsensor 设备使用示例说明

在目前使用的输入设备中，sensor 类设备都没有 gpio 的申请，只需要获取配置参数即可。以 bma250 为例子进行说明，如下：

```
......
static struct sensor_config_info config_info = {//第一步
    .input_type = GSENSOR_TYPE,
};
......
static int __init BMA250_init(void)
{
......
    if(input_fetch_sysconfig_para(&(config_info.input_type))){//第二步
        printk("%s: err.\n", __func__);
        return -1;
```

```
    }
    if(config_info.sensor_used == 0){//第三步
        printk("*** used set to 0 !\n");
        printk("*** if use sensor,please put the sys_config.fex gsensor_used set to 1.
\n");
        return 0;
    }
     ......
}
```

步骤如下：

第一步：定义 sensor_config_info 结构体，并且将 input_type 赋值为 GSENSOR_TYPE

第二步：init 函数中调用 input_fetch_sysconfig_para 函数获取 sysconfig.fex 下的配置参数。

第三部：判断获取到的 xxx_used 是否为 1，若不为 1，则直接返回。

与 ctp 的相比即少了 gpio 资源的申请。

5. Declaration

This document is the original work and copyrighted property of Allwinner Technology ("Allwinner"). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner.

The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This datasheet neither states nor implies warranty of any kind, including fitness for any particular application.