全 志 科 技
Allwinner Technology

A20 平台 CTP 模块开发说明文档

V2.0

2013-06-17

Revision History

Version	Date	Changes compared to previous issue
V1.0	2013-03-15	初始版本
V2.0	2013-06-17	1. 删除了 **init-ctp** 相关文件的接口说明 2. 修改了 **ctp** 模块中读取 **sysconfig.fex** 文件的接口 3. 修改了 **gt** 系列，**gsl** 系列模组的使用方法 4. 增加了 **ctp** 自动检测的一些配置项

目录

1.概述

1.1.编写目的

文档对 CTP 硬件以及软件相关的调试与移植过程作详细的讲解，同时对 CTP 的使用与测试作简要的讲解，为达到能快速移植驱动与使用 CTP 的目的。

1.2.使用范围

适用于 A20 对应平台。

1.3.相关人员

项目中 ctp 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

第二章主要讲解 CTP 模块的结构以及模块的硬件连接、需要关注的关键点，源码路径以及模块的相关配置。第三章主要讲解 CTP 中平台相关接口以及硬件资源的申请。第四章对驱动移植中为适应平台需要修改的地方进行说明。第五章主要讲解 CTp 的调试步骤。第六章主要讲解平台中 CTP 的配置，即如何使用一款 CTP。第七章主要 CTP 的测试，以及测试中问题点的定位。（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）

2.模块介绍

2.1.模块功能介绍

触摸（CTP）驱动作为硬件与软件的一个桥梁，实现对触摸控制器硬件初始化，接收触摸控制器发送的位置坐标信息（必要时，对数据进行滤波和动作识别处理），上报用户手势相关信息于操作系统，经上层系统处理后，正确响应触摸者的意图。

CTP 属于 INPUT 设备，因此 CTP 驱动设计必须符合整个系统的要求以及 INPUT 系统设计规范。CTP 使用的通信接口可能为 USB，SPI，I2C 等，因此 CTP 驱动设计时应符合相关通信接口的设计规范。目前使用的 CTP 均为使用 i2c 通信接口。

2.2.模块体系结构描述

模块体系结构如下图所示：

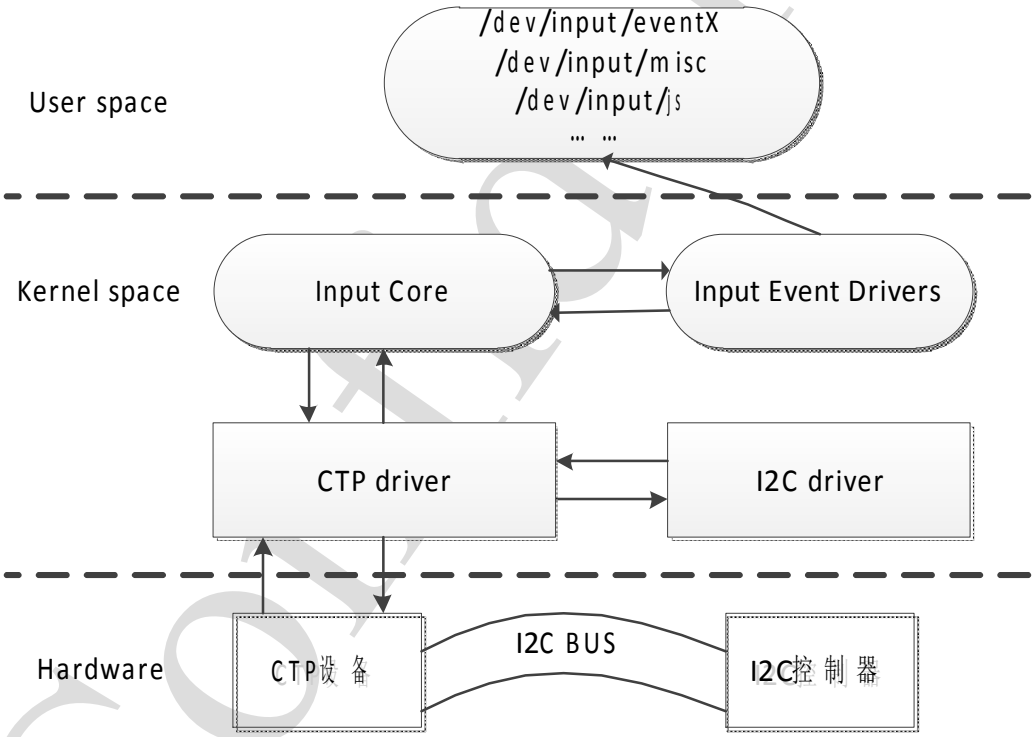

图 1 模块体系结构图

CTP 设备为使用 I2C 总线进行通信的输入设备，需调用到 Linux 的 Input 子系统中的相应接口进行设备的注册，同时需要调用 I2C 总线的设备进行 I2C 设备的注册以便进行数据的通信。

CTP Driver 提供对硬件各寄存器的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件，再通过核心层（Input Core）提交给事件处理层；而核心层对下提供了 CTP Driver 的编程接口，对上又提供了事件处理层的编程接口；而事件处理层（input Event Driver）就为我们用户空间的应用程序提供了统一访问设备的接口和驱动

层提交来的事件处理。用户空间将根据设备的节点进行数据的读取以及相应的处理。

2.3.模块硬件介绍

目前 CTP 与 HOST 的连接主要有 6 个 pin 脚，分别为 VCC，GND，INT，WAKE_UP(RESET),SDA,SCL。引脚正常工作时候的高电平均为 3.3V。

硬件电路连接如下图所示：

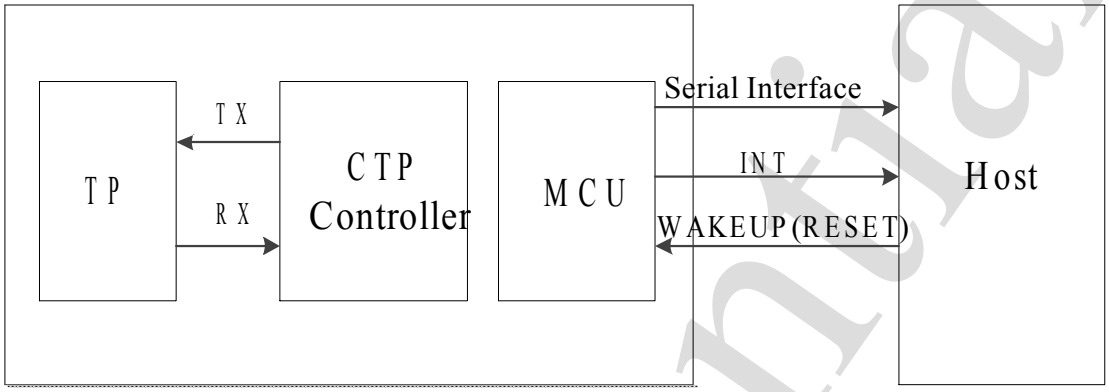

图 2 ctp 硬件连接图

CTP 的调试中，首先确认硬件的正确性。硬件调试中，需要确认下列项：

1）确认各个引脚与 HOST 连接的正确性。

2）电源电压是不是正常的，VCC 是否为 3.3V，GND 是否为 0。

3）I2C 引脚电平是否匹配。

4）复位电路是否正常，主要检查 WAKE_UP(RESET)引脚的电压是否正常。

5）INT 引脚电压是否正常。

6）确认设备使用的 I2C 地址，特别是设备可以设置为多个地址时。

2.4.源码结构介绍

CTP 初始化部分源码位置如下：

我司平台移植的 ctp 驱动均放置在一个统一的目录下面，CTP 模块驱动的源码位置如下：\lichee\linux-3.4\drivers\input\sw_touchscreen

FT5x（ft5x_ts.c）系列，gsl(gslX680.c)系列，gt(gt82x.c)等系列的源码均位于该目录下。

2.5.模块配置介绍

2.5.1. sys_config.fex 配置

（1）ctp 使用配置

配置文件的位置：\lichee\tools\pack\chips\sun7i\configs\android\wing-XXX 目录下。

配置文件 sys_config.fex 中关于 CTP 的配置项如下：

```
[ctp_para]
ctp_used                = 1
ctp_twi_id              = 2
ctp_name                = "gt813_evb"
ctp_screen_max_x        = 1280
ctp_screen_max_y        = 800
ctp_revert_x_flag       = 1
ctp_revert_y_flag       = 1
ctp_exchange_x_y_flag   = 1
ctp_int_port            = port:PH21<6><default><default><default>
ctp_wakeup              = port:PB13<1><default><default><1>
```

文件配置说明如下：

ctp_twi_id	使用哪组 I2C
ctp_name	设备的名字，用于同一驱动中，区别不同型号的 TP 面板参数
ctp_twi_addr	I2C 设备地址（7 位地址）
ctp_screen_max_x ， ctp_screen_max_y	touch panel 的分辨率
ctp_revert_x_flag	如果 x 轴反向，请置为 1
ctp_revert_y_flag	如果 y 轴反向，请置为 1
ctp_exchange_x_y_flag	如果 x、y 倒置，请置为 1
ctp_int_port	触摸中断引脚
ctp_wakeup	触摸 wakeup 引脚
ctp_io_port	中断引脚配置

触摸模块中的 GPIO 需要正确的配置中断引脚、wakeup（reset），根据平台中使用的引脚进行相应的配置。

（2）**ctp 自动检测配置**

```
[ctp_list_para]
ctp_det_used        = 1 //设置为 1 时，启动自动检测，设置为 0 时，退出自动检测。
ft5x_ts             = 1 //设置为 1，该模块支持的 I2C 地址添加到扫描列表
gt82x               = 0 //设置为 0，该模块支持的 I2C 地址从扫描列表中剔除
gslX680             = 1
gt9xx_ts            = 1
gt811               = 1
zet622x             =1
```

ctp_list_para列表中的名称顺序必须与 sw_device.c中ctps的名称顺序一一对应。

当ctp_det_used 设置为1时，启用自动检测，将设置为0时，退出自动检测。模块的名称后面写1表示添加到自动检测扫描列表，写0表示剔除自动检测扫描列表。

2.5.2. menuconfig 的配置

在编译服务器上，目录为\lichee\linux-3.4 上，输入命令：

```
make ARCH=arm menuconfig
```

如下所示：

```
/linux-3.4$ make ARCH=arm menuconfig
```

进入目录 Device Drivers\Input device support\Touchscreens 目录下可以看到 ctp 模块是编译为模块、编译进内核、不编译。为了便于配置以及更换设备，ctp 驱动均编译为模块的形式。

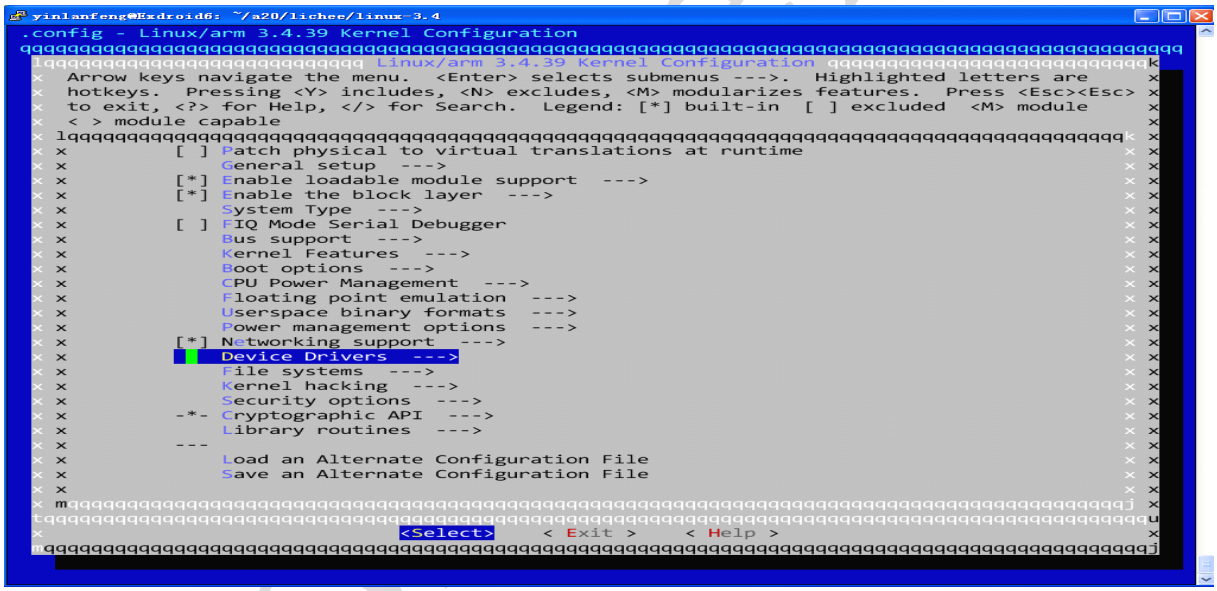

图 3　Device Drivers 选项配置

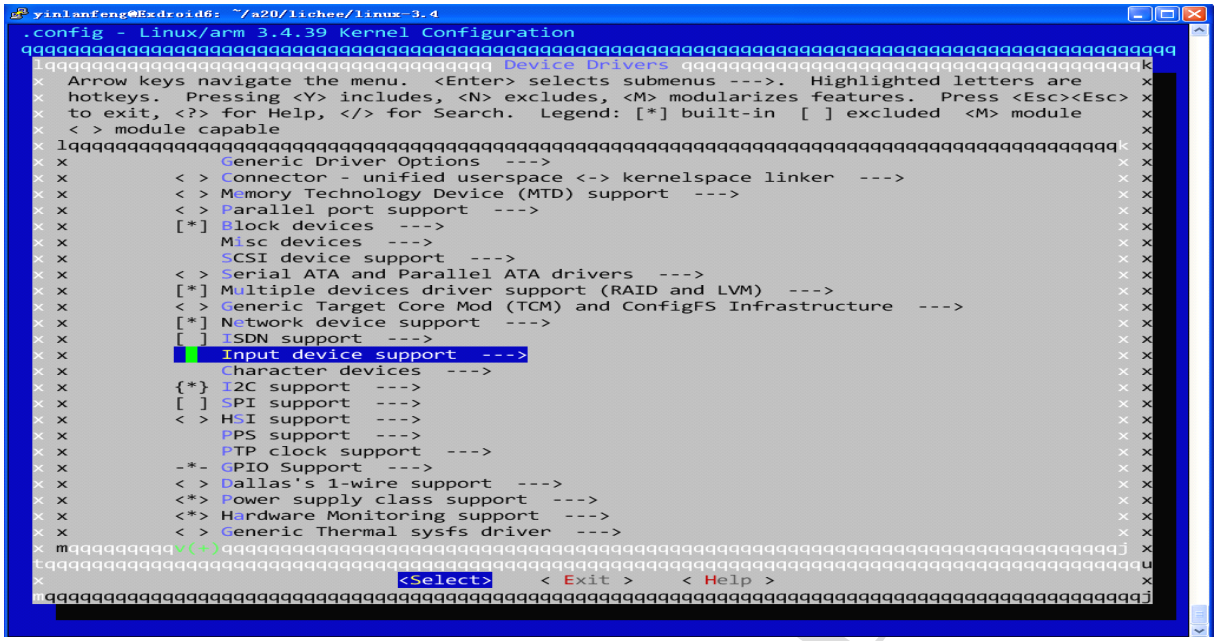

图 4 Input device support 选项配置

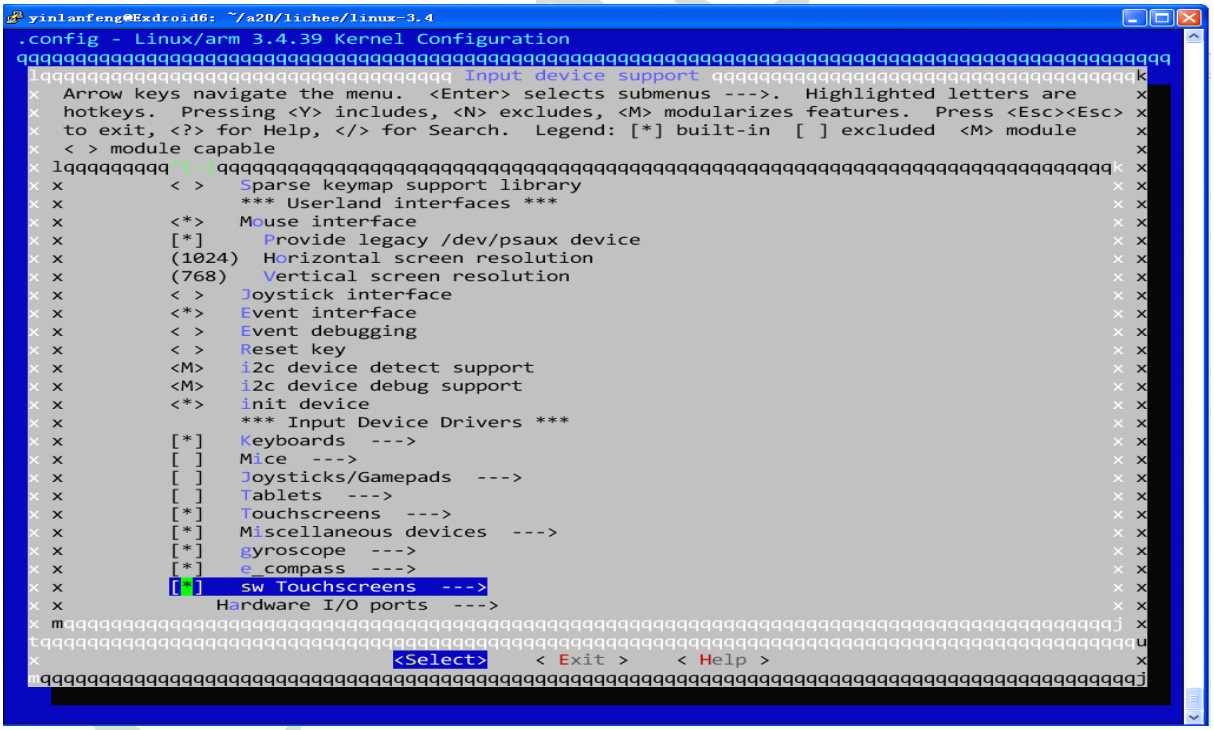

图 5 sw_touchscreens 配置选项

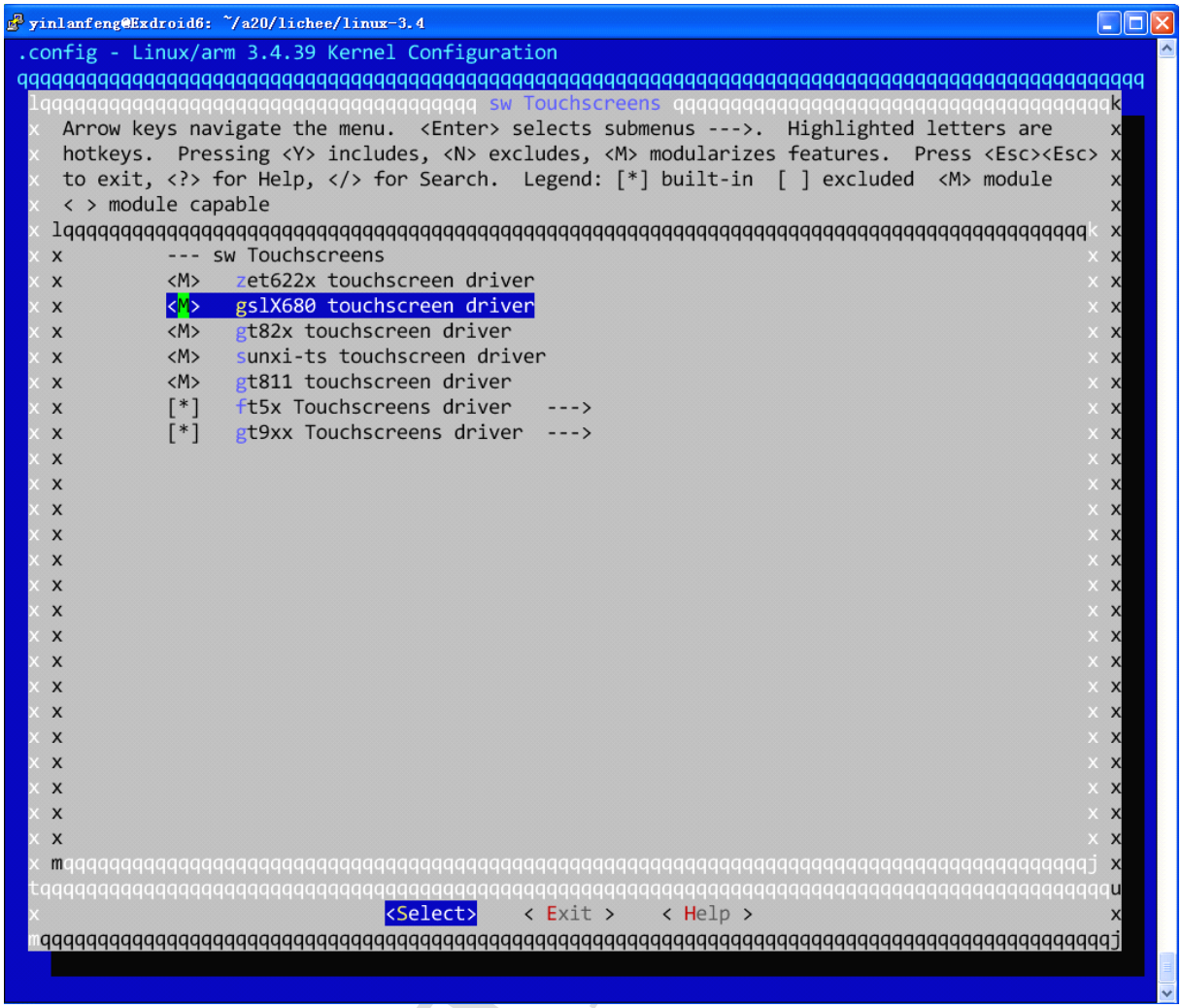

图 6 gslX680 驱动配置

3.CTP 驱动架构图

平台中，将系统使用的硬件资源、配置文件与模组的驱动分开，CTP 驱动的结构图如下所示：

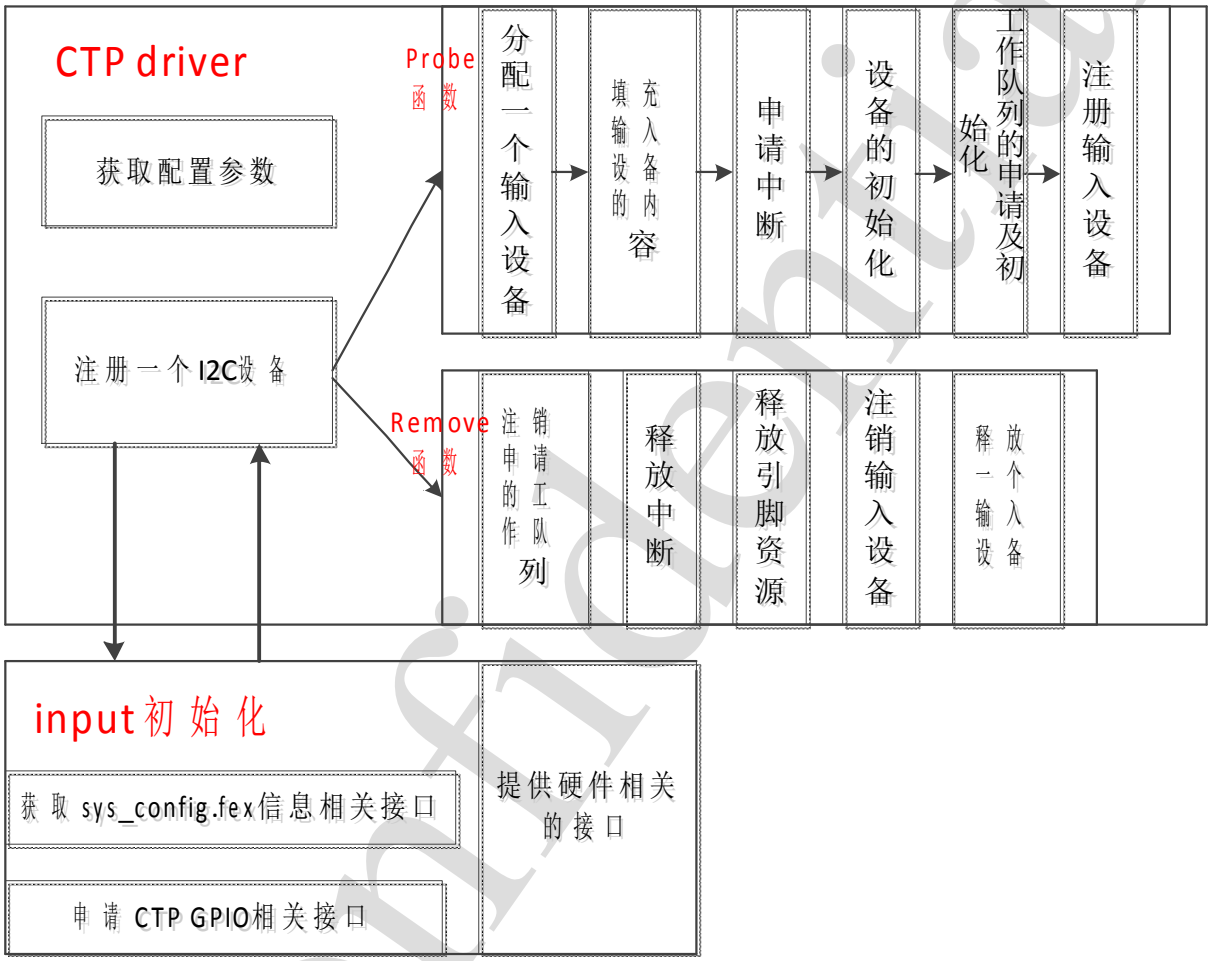

图 7 ctp 驱动结构图

input 初始化中，主要工作是提供获取 sys_config.fex 文件中的配置信息，申请使用的 GPIO 引脚，封装系统函数对中断的使能，频率的设置等与硬件相关信息等相关接口。模组的驱动中（即 CTP driver）根据第四章中驱动的移植进行相关函数的修改即可完成驱动的移植工作，无需添加相应的硬件相关接口。

接下来将详细介绍 CTP 驱动初始化的接口以及关键数据结构，在第四章中将详细介绍 CTP 驱动移植的步骤。

input 初始化包含获取种设备的 sysconfig.fex 的配置参数以及 ctp 相关的接口，源码位于：\lichee\linux-3.4\drivers\input\sw_device.c

4. CTP 驱动移植

驱动的移植中主要需要关注点为如何获取 sys_config.fex 中的配置信息，中断的申请，detect 函数，suspend 以及 resume 函数等。以下将以 ft5x 系列驱动移植过程为例进行说明。源码路径为：\lichee\linux-3.4\drivers\input\sw_touchscreen
ft5x\ft5x.c

4.1. 驱动中 INPUT 子系统关键接口

这部分接口是 linux INPUT 子系统对外提供的接口，tp driver 使用这些接口，向 input 子系统注册设备和上报数据。

（1）申请 input_dev 结构：

```
input_dev = input_allocate_device();
```

（2）注册输入设备，并和对应的 handler 处理函数挂钩：

```
err = input_register_device(input_dev);
```

（3）注销 Input 设备：

```
input_unregister_device(ft5x_ts->input_dev);
```

（4）上报坐标值（绝对值）：

```
static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value)
```

（5）上报多点坐标值时多点同步信号：

```
static inline void input_mt_sync(struct input_dev *dev)
```

（6）上报坐标值结束时的同步信号：

```
static inline void input_sync(struct input_dev *dev)
```

4.2. I2C 设备关键接口

目前 CTP 设备驱动使用的都是 I2C 总线进行通信，关键的接口如下所示：

（1）、i2c_set_clientdata

将设备驱动的私有数据连接到设备 client 中。

（2）、i2c_get_clientdata

获取设备 client 的设备驱动的私有数据。

（3）、i2c_add_driver

通过 I2C 核心的 i2c_add_driver()函数添加 i2c_driver,使用到的关键数据结构为 i2c_driver,注意 i2c_driver 中使用的 name 需与 INPUT 中的 input_dev name 一致，否则有会出现找不到设备的情况。结构体定义如下所示：

```
static struct i2c_driver ft5x_ts_driver = {
        .class = I2C_CLASS_HWMON,
        .probe          = ft5x_ts_probe,
        .remove         = __devexit_p(ft5x_ts_remove),
        .id_table       = ft5x_ts_id,
        .driver  = {
                .name   = CTP_NAME,
                .owner  = THIS_MODULE,
        },
        .address_list   = normal_i2c,
};
```

（4）、i2c_del_driver

通过 I2C 核心的 i2c_del_driver()函数删除 i2c_driver。

4.3.驱动需要包含的头文件

ctp 驱动中需要包含头文件：

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

这个头文件中包含使用的 struct ctp_config_info 结构体以及读取 sysconfig.fex 的相关调用接口，申请 gpio 相关接口、ctp int 引脚的相关硬件配置接口、打印等级的定义。

该文件的具体实现:\lichee\linux-3.4\drivers\input\sw_device.c

可查阅文档“A20 平台 input 初始化文档.doc”

4.4.驱动关键数据结构

ctp_config_info,该数据结构主要存放从 sys_config.fex 中获取的信息，如 tp 的分辨率，使用的 wakeup 引脚，irq 引脚等。数据结构的定义如下：

```
struct ctp_config_info{
    enum input_sensor_type   input_type; //设备类型，ctp 的应该为 CTP_TYPE
    int       ctp_used;//是否使用 ctp，当该变量为 0 时，则结束初始化过程
    __u32     twi_id;//使用的 I2C 组别号，为，0,1,2,3 等数值
    int       screen_max_x;//Tp X 轴最大值
    int       screen_max_y;//Tp X 轴最大值
    int       revert_x_flag;//X 轴反转标志
    int       revert_y_flag;//  Y 轴反转标志
    int       exchange_x_y_flag;//X，Y 轴互换标志
    u32       int_number;//Irq 使用的 gpio 号
    u32       wakeup_number;//Wakeup(reset)使用的 gpio 号
```

```
#ifdef TOUCH_KEY_LIGHT_SUPPORT
        u32     light_number; //有按键灯时，使用的 gpio 号
#endif
};
```

4.5.设备驱动关键变量

移植新的设备驱动时，需要设置一些跟平台相关的关键的变量，如表所示：

名称	含义	使用位置
ctp_debug	设置打印等级变量	在 module_para_name 中使用
config_info	存取 sys_config.fex 获得的配置值	主要用于 ctp_get_system_config 函数中，获取配置值
CTP_IRQ_NUMBER	IRQ 的 GPIO 引脚号，从 config_info 中获取	用于中断申请 sw_gpio_irq_request 等中断操作的相关函数中，
CTP_IRQ_MODE	IRQ 中断的触发方式	用于中断申请 sw_gpio_irq_request
CTP_NAME	设备模组名称	用于设备的注册等
SCREEN_MAX_X（screen_max_x）	X 轴最大分辨率	用于上报数值的控制，以及数据读取时处理等。
SCREEN_MAX_Y (screen_max_y)	Y 轴最大分辨率	用于上报数值的控制，以及数据读取时处理等。
revert_x_flag	X 轴反转标志	当 X 轴反置时使用，用于数据的处理或者是上报函数中
revert_y_flag	Y 轴反转标志	当 Y 轴反置时使用，用于数据的处理或者是上报函数中
exchange_x_y_flag	X，Y 轴互换标志	当 X，Y 轴需要互换时使用，用于数据的处理或者是上报函数中
int_handle	中断申请句柄	为 sw_gpio_irq_request 的返回值，用于 sw_gpio_irq_free 中断释放函数中
normal_i2c	设备地址数组	i2c_driver 结构体中，注册设备地址
chip_id_value	Chip ID 值数组	当 detect 函数中使用硬件检测时使用
twi_id	I2C 组别号	用于 detect 函数中

4.6. Kconfig 以 Makefile 文件

当添加一个设备模组时，需要修改该目录下的 Kconfig 以及 Makefile 文件，使得能够在 menuconfig 中选设备模组驱动并编译生成模块或者是直接编译进内核。如下所示：

Kconfig 中需要添加的信息：

```
config SW_TOUCHSCREEN_FT5X_TS
        tristate    "ft5x touchscreen driver"
        default m
        depends on INPUT && I2C
        help
          ft5x touchscreen driver
```

Makefile 文件中需要添加的信息：

```
obj-$(CONFIG_SW_TOUCHSCREEN_FT5X_TS)          += ft5x_ts.o
ft5x_ts-objs := ft5x02_config.o ini.o ft5x.o
```

Ft5x 驱动兼容 ft5x06 以及 ft5x02，因此需要添加 ft5x02 的相关文件，为多文件编译为一个 KO 文件，注意格式的写法。

Kconfig 中 config 后的 SW_TOUCHSCREEN_FT5X_TS 与 Makefile 后 CONFIG_后的 SW_TOUCHSCREEN_FT5X_TS 名称必须相同。一般情况下以模组的名称命名，便于识别。

添加之后可以到系统中查看是否添加成功，在编译服务器上，目录为\lichee\linux-3.4 上，输入命令：

```
make ARCH=arm menuconfig
```

进入目录 Device Drivers\Input device support\Touchscreens 目录下查看是否存在添加的模组的选项。

4.7. I2C 地址

设备的 I2C 地址以固定的形式存在于设备驱动中。存放设备地址的数组必须以 I2C_CLIENT_END 标致结束。如下所示:

```
static const unsigned short normal_i2c[2] = {0x38,I2C_CLIENT_END};
```

4.8. detect 函数

detect 函数为硬件设备的检测，一般情况下通过读取 IC 的 chip ID 号或者是检测 I2C 是否通信成功进行检测。当 chip ID 不匹配，或者是 I2C 通信不成功时，将返回失败，则注册 I2C 设备失败。因此当使用 I2C 通信以及读取设备 chip ID 时，必须保证在 detect 函数中时，设备已经准备就绪，在硬件没有错误的情况下能进行正常的 I2C 通信过程。也可以不检测，但在后期设备的通信中无法正常保证设备通信正常。

4.8.1.读取 chip ID

读取 chip ID 值，以 ft5x.c 中的 ctp_detect 函数为例，如下：

```
static int ctp_detect(struct i2c_client *client, struct i2c_board_info *info)
```

```
{
    struct i2c_adapter *adapter = client->adapter;
        int ret = 0, i = 0;
        if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_BYTE_DATA))
                return -ENODEV;

    if(twi_id == adapter->nr){
            ret = i2c_smbus_read_byte_data(client,0xA3);
        if(ret == -70) {
            msleep(10);
            ret = i2c_smbus_read_byte_data(client,0xA3);
        }
    dprintk(DEBUG_INIT,"addr:0x%x,chip_id_value:0x%x\n",client->addr,ret);
                while(chip_id_value[i++]){
                        if(ret == chip_id_value[i - 1]){
                                strlcpy(info->type, CTP_NAME, I2C_NAME_SIZE);
                chip_id = ret;
                            return 0;
                        }
                }
            printk("%s:I2C connection might be something wrong ! \n",__func__);
            return -ENODEV;
    }else{
        return -ENODEV;
    }
}
```

使用该方法时，必须确保读取出来的 chip ID 值，已经存放在 chip_id_value 数组中，否则将造成检测失败的情况。

4.8.2. I2C 通信的方式

I2C 通信的方式进行检测，以 gslX680.c 中的 ctp_detect 函数为例，ci2c_test 函数存放在 init-input.c 中。如下：

```
static int ctp_detect(struct i2c_client *client, struct i2c_board_info *info)
{
    struct i2c_adapter *adapter = client->adapter;
    int ret;

        if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_BYTE_DATA))
                return -ENODEV;

    if(twi_id == adapter->nr){
                dprintk(DEBUG_INIT,"%s: addr= %x\n",__func__,client->addr);
                ret = i2c_test(client);
                if(!ret){
            printk("%s:I2C connection might be something wrong \n",__func__);
            return -ENODEV;
        }else{
            strlcpy(info->type, CTP_NAME, I2C_NAME_SIZE);
            return 0;
              }

    }else{
        return -ENODEV;
    }
}
```

4.8.3.不使用硬件检测的方式

不使用硬件检测时，将直接加载设备的驱动，无法判断硬件设备是否存在，可能会造成加载错误等情况发生，因此建议使用硬件加载的方式。

不使用硬件检测的方式时，如下所示：

```
static int ctp_detect(struct i2c_client *client, struct i2c_board_info *info)
{
    struct i2c_adapter *adapter = client->adapter;
    if(twi_id == adapter->nr){
```

```
            dprintk(DEBUG_INIT,"%s: addr= %x\n",__func__,client->addr);
            strlcpy(info->type, CTP_NAME, I2C_NAME_SIZE);
            return 0;
    }else{
        return -ENODEV;
    }
}
```

4.9. sys_config.fex 参数的获取

增加模组时，需要增加 ctp_get_system_config 函数获取系统的配置参数，当读取不到时，返回失败。该函数如下所示：

```
static int ctp_get_system_config(void)
{
        twi_id = config_info.twi_id;
        screen_max_x = config_info.screen_max_x;
        screen_max_y = config_info.screen_max_y;
        revert_x_flag = config_info.revert_x_flag;
        revert_y_flag = config_info.revert_y_flag;
        exchange_x_y_flag = config_info.exchange_x_y_flag;
        if((twi_id == 0) || (screen_max_x == 0) || (screen_max_y == 0)){
                printk("%s:read config error!\n",__func__);
                return 0;
        }
        return 1;
}
```

4.10. 中断申请

4.10.1 中断申请函数介绍

平台中将封装了中断的申请函数，为 sw_gpio_irq_request。

u32 sw_gpio_irq_request(u32 gpio, enum gpio_eint_trigtype trig_type,
peint_handle handle, void *para)

功能: 申请 gpio 中断.
参数: gpio: gpio 编号.
trig_type: 触发类型.

```
enum gpio_eint_trigtype {
        TRIG_EDGE_POSITIVE = 0,
        TRIG_EDGE_NEGATIVE,
        TRIG_LEVL_HIGH,
        TRIG_LEVL_LOW,
        TRIG_EDGE_DOUBLE,       /* positive/negative */
        TRIG_INALID
};
```

Handle: 中断回调函数. 当 gpio 中断触发时回调.
Para: handle 的参数. 必须为全局, 或者在堆中, 不能为栈中的局部变量.

返回值: 成功返回句柄, 失败返回 0.
该函数内部会进行的操作为:
(1) 检测 gpio 是否可配置成中断
(2) 配置 gpio 的功能(mul sel), pull, driver level, trig type, 并打开 gpio 中断的 enable 位. 这些通过调用 sw_gpio_eint_setall_range 来完成.
(3) 分配 gpio 中断句柄(gpio_irq_handle 结构)作为返回值
(4) 向 linux 内核申请中断. 调用 request_irq, 并传入 IRQF_SHARED 标记。

4.10.2 中断释放函数介绍

释放 IRQ 时, 使用的函数为 sw_gpio_irq_free.
u32 sw_gpio_irq_free(u32 handle)
功能: 释放 gpio 中断. 与 sw_gpio_irq_request 对应.
参数: Handle: sw_gpio_irq_request 返回的句柄.
返回值: 成功返回 0, 失败返回错误行号.
该函数会进行的操作:
(1) 关闭 gpio 中断的 enable 位. 调用 sw_gpio_eint_set_enable.
(2) 清 gpio 中断的 pending 位. 调用 sw_gpio_eint_clr_irqpd_sta.
(3) 向 linux 内核释放 gpio 中断. 调用 free_irq.
(4) 释放 gpio 中断句柄. 即 sw_gpio_irq_request 返回的句柄.
(5) 释放 gpio 的访问权. 调用 gpio_free.

4.10.3 中断操作例子

因此在模组驱动中,只需要调用 sw_gpio_irq_request 函数即可完成中断的申请操作。

如下所示：

```
int_handle =
sw_gpio_irq_request(CTP_IRQ_NUMBER,CTP_IRQ_MODE,(peint_handle)ft5x_ts_interrupt,ft5x_ts);
        if(!int_handle) {
                printk("ft5x_ts_probe: request irq failed\n");
                goto exit_irq_request_failed;
        }
```

中断服务程序也跟之前有所差别，平台中会根据中断申请进行相应的处理，并进行清中断操作，修改后的中断服务程序返回值为 0 表示成功，其他值表示错误。如下所示：

```
static u32 ft5x_ts_interrupt(struct ft5x_ts_data *ft5x_ts)
{
        dprintk(DEBUG_INT_INFO,"==========ft5x_ts TS Interrupt===========\n");
        queue_work(ft5x_ts->ts_workqueue, &ft5x_ts->pen_event_work);
        return 0;
}
```

中断函数的释放，当退出驱动，或者是不在使用时，需要释放 IRQ，int_handle 为申请中断时的标记。如下所示：

```
static int __devexit ft5x_ts_remove(struct i2c_client *client)
{
...  ...
            sw_gpio_irq_free(int_handle);
...  ...


}
```

4.11. super standby 支持

Suspend 分为 early suspend 与 suspend ，resume 分为 late resume 与 resume。Early suspend 与 late resume 为一对，当定义 CONFIG_HAS_EARLYSUSPEND 使用；suspend 与 resume 为一对，该接口主要定义于 i2c_driver 结构体中。以上四个函数的接口的关系要处理好，否则可能导致无法唤醒等意外情况发生。在 suspend 与 early suspend 函数中，需要清读取数据的队列，防止工作队列中的工作还没有完成，但是硬件已经处于不工作状态。在 early susupend 与 later resume 为一对，在 early suspend 函数中，做了停止的操作，在 late resume 函数中就需要做相对应的恢复工作。Suspend 与 resume 函数也是如此。

当 sys_config.fex 中 standby_mode 设置为 1 时表示支持 super standby，super standby 就是关掉除 AVCC 和 DRAM_VCC 电源以外的所有电源；这样的操作有可能改变模块状态。因此当开启 super standby 时，驱动中的 suspend 以及 resume 函数需要做相应的操作，支持 super standby。防止开始 super standby 时，机器唤醒之后会出现触摸失效的现象。

需要做的步骤如下所示：

（1）包含头文件 linux/pm.h；

（2）判断 standby 类型,并进行相应处理；内核导出了两个符号：standby_type, standby_level 以利于各模块：根据目标区分 normal standby 和 super standby。根据掉电情况，区别对待设备；

standby_type：表目标，支持 NORMAL_STANDBY，SUPER_STANDBY；

standby_level ： 表 结 果 ， 支 持 STANDBY_WITH_POWER_OFF ， STANDBY_WITH_POWER；

如果设备在 norma standby 以及 super standby 情况需要做的处理是一样的也可以不进行区分。在掉电的情况下，有的设备需要重新写配置参数的，需要进行相应的区分，以保证能快速的唤醒，且在休眠设备掉电后，重新唤醒时，设备也能正常稳定的使用。Suspend 函数中需要将硬件以及软件的动作停止掉，在写寄存器等操作让设备进入休眠。如下所示：

```
static void ft5x_ts_suspend(struct early_suspend *handler)
{
    dprintk(DEBUG_SUSPEND,"==ft5x_ts_suspend=\n");
    dprintk(DEBUG_SUSPEND,"ft5x_ts_suspend: write FT5X0X_REG_PMODE.\n");
    sw_gpio_eint_set_enable(CTP_IRQ_NUMBER,0);
    cancel_work_sync(&ft5x_ts->pen_event_work);
    ft5x_set_reg(FT5X0X_REG_PMODE, PMODE_HIBERNATE);

}
```

Resume 函数中则进行相关的动作，如下所示：

```
static void ft5x_ts_resume(struct early_suspend *handler)
{
        dprintk(DEBUG_SUSPEND,"==ft5x_ts_resume== \n");
        ctp_wakeup(0,20);
        sw_gpio_eint_set_enable(CTP_IRQ_NUMBER,1);
        if(STANDBY_WITH_POWER_OFF == standby_level){
            msleep(100);
        }
}
```

4.12.设备驱动 init 函数

该函数中主要任务为调用相关函数获取 sys_config.fex 中的配置信息，以及注册 i2c 设备驱动。如下所示：

```
static int __init ft5x_ts_init(void)
{
    ......
    if (input_fetch_sysconfig_para(&(config_info.input_type))) { //获取配置参数
```

```
        printk("%s: ctp_fetch_sysconfig_para err.\n", __func__);
        return 0;
    } else {
        ret = input_init_platform_resource(&(config_info.input_type));//申请 ctp 引脚资源
        if (0 != ret) {
            printk("%s:ctp_ops.init_platform_resource err. \n", __func__);
        }
    }
    if(config_info.ctp_used == 0){
            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;
    }
        if(!ctp_get_system_config()){
                printk("%s:read config fail!\n",__func__);
                return ret;
        }
    ctp_wakeup(config_info.wakeup_number, 0, 10);
    ft5x_ts_driver.detect = ctp_detect;
        ......
        ret = i2c_add_driver(&ft5x_ts_driver);
        ......
}
```

4.13.设备驱动 exit 函数

exit 函数主要是卸载 i2c 驱动，以及退出申请的 ctp 硬件资源等。

```
static void __exit ft5x_ts_exit(void)
{
    printk("==ft5x_ts_exit==\n");
    i2c_del_driver(&ft5x_ts_driver);
    class_destroy(i2c_dev_class);
    unregister_chrdev(I2C_MAJOR, "aw_i2c_ts");
```

```
    input_free_platform_resource(&(config_info.input_type));
}
```

4.14.设备驱动 remove 函数

模块卸载时，注意 probe 函数与 init 函数中申请的资源，要依照申请的顺序进行释放，后申请的先释放。如果申请的资源没有释放，或没有按照顺序释放，在模块卸载时，会卸载失败，甚至导致死机。

同时，在卸载时，注意要调用 i2c_set_clientdata(client, NULL);函数。否则驱动自己是可以正常卸载的，但卸载后，向此 I2C 加载其它 gsensor 驱动时会出现 I2C 通讯不成功问题。

4.15.ctp_wakeup 函数

ctp_wakeup 函数的实现在 init-input.c 中，主要作用为设置 ctp wakeup 引脚的输出电平以及延时的时间等。函数的接口如下所示：

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

gpio: 为 wakeup 引脚的 gpio 编号，存在于 struct ctp_config_info 结构体的变量 wakeup_number 中。

status：为输出的 gpio 的状态，0 表示低电平，1 表示高电平。

ms：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。
```

4.16.驱动调试信息

驱动的设计中，通过模块的参数来设置相关的打印信息。默认情况下，打印信息为不开启的，主要用于调试使用。为了不需要重新编译，导入等一些麻烦操作，所以在驱动中添加模块的参数来设置相关的打印信息。打印信息的等级定义在 init-input.h 中，当将模块参数中设置打印信息的开关打开时，将打印相关的打印信息。打印开关的设置如下所示：

```
enum {
    DEBUG_INIT = 1U << 0,
    DEBUG_SUSPEND = 1U << 1,
    DEBUG_INT_INFO = 1U << 2,
    DEBUG_X_Y_INFO = 1U << 3,
    DEBUG_KEY_INFO = 1U << 4,
    DEBUG_WAKEUP_INFO = 1U << 5,
    DEBUG_OTHERS_INFO = 1U << 6,

};
```

表 3 打印相关等级的含义

名称	含义
DEBUG_INIT	Init 打印信息
DEBUG_SUSPEND	休眠，唤醒相关打印信息
DEBUG_INT_INFO	中断相关打印信息
DEBUG_X_Y_INFO	X，Y 坐标值相关打印信息
DEBUG_KEY_INFO	按键相关打印信息
DEBUG_WAKEUP_INFO	Wakeup 引脚相关打印信息
DEBUG_OTHERS_INFO	其他相关打印信息

驱动中需要增加模块参数的设计，以及更换模块的打印信息语句。增加模块的参数如下所示，默认情况下，将 DEBUG_INIT 的打印信息打开：

```
…… ……
static u32 ctp_debug = DEBUG_INIT;
#define dprintk(level_mask,fmt,arg...)    if(unlikely(ctp_debug & level_mask)) \
    printk("***CTP***"fmt, ##arg)
…… ……
module_param_named(ctp_debug,ctp_debug,int,S_IRUGO | S_IWUSR | S_IWGRP);
…… ……
```

打印语句的修改如下所示：

```
…… ……
dprintk(DEBUG_SUSPEND,"==ft5x_ts_suspend=\n");
dprintk(DEBUG_SUSPEND,"ft5x_ts_suspend: write FT5X0X_REG_PMODE.\n");
…… ……
dprintk(DEBUG_X_Y_INFO,"report:===x5 = %d,y5 = %d ====\n",event->x5,event->y5);
…… ……
```

为了防止驱动加载失败等情况时，没有城市模块的节点，无法定位设备驱动的运行情况。因此设备驱动中的 init，probe 函数中的打印信息还是使用 printk 进行输出。

4.17.模块加载及 resume 延时优化

ctp 模块的加载以及 resume 的过程中，以下几点将加长模块的加载以及唤醒时间。

（1）针对相对应的 tp 屏，进行相对应参数的下载。

（2）为使 tp 能稳定正常工作，进行相应的延时

（3）操作 i2c 总线，进行读写操作。

以上三点中第一与第二点耗时最多，模块的加载时间，最高应该控制在50个毫秒之内，若不进行优化，模块的加载最高可达到几个秒，无形中延长了开机时间。

针对参数的下载，延时，i2c 总线的操作等耗时的工作，将其放到队列中工作，这样不但加快了模块的加载时间，同时还能按照上电时序进行相对应的操作，何乐而不为。具体的操作可以参照 gslX680.c 中的做法，该驱动中，没有进行优化前，下载参数需要 2S 左右，提高 i2c 速度后，也需要600多毫秒，优化后，模块的加载时间为20个毫秒内。

以下以 gslX680.c 中的优化进行相关的说明。队列的定义如下所示：

```
static void gslX680_init_events(struct work_struct *work);
static void gslX680_resume_events(struct work_struct *work);
struct workqueue_struct *gslX680_wq;
struct workqueue_struct *gslX680_resume_wq;
static DECLARE_WORK(gslX680_init_work, gslX680_init_events);
static DECLARE_WORK(gslX680_resume_work, gslX680_resume_events);
```

队列的创建：

```
gslX680_wq = create_singlethread_workqueue("gslX680_init");
        if (gslX680_wq == NULL) {
                printk("create gslX680_wq fail!\n");
                return -ENOMEM;
        }
gslX680_resume_wq = create_singlethread_workqueue("gslX680_resume");
        if (gslX680_resume_wq == NULL) {
                printk("create gslX680_resume_wq fail!\n");
                return -ENOMEM;
        }
```

工作队列中需要做的工作，如下：

```
static void gslX680_resume_events (struct work_struct *work)
{
        ctp_wakeup(1,5);
        ctp_wakeup(1,0);
        msleep(10);
    reset_chip(gslX680_i2c);
    startup_chip(gslX680_i2c);
    check_mem_data(gslX680_i2c);
    sw_gpio_eint_set_enable(CTP_IRQ_NUMBER,1);

}
.....
static void gslX680_init_events (struct work_struct *work)
{
        gslX680_chip_init();
        init_chip(gslX680_i2c);
        check_mem_data(gslX680_i2c);
        int_handle =
sw_gpio_irq_request(CTP_IRQ_NUMBER,CTP_IRQ_MODE,(peint_handle)gsl_ts_irq,ts_init);
        if(!int_handle) {
                printk( "gsl_probe: request irq failed\n");
        }
        return;
}
```

队列的使用，init 队列主要用于 probe 函数中，resume 队列用于唤醒函数中。

```
static int gsl_ts_resume(struct i2c_client *client)
{
.....
queue_work(gslX680_resume_wq, &gslX680_resume_work);
.....
}
.....
static int __devinit gsl_ts_probe(struct i2c_client *client,
                                  const struct i2c_device_id *id)
{
.....
queue_work(gslX680_wq, &gslX680_init_work);
.....
}
```

使用队列的注意事项：

由于队列不在主线程中进行，其工作的时间不可预估，因此，队列中的工作应该跟时序无关，或者队列中的整个工作过程必须满足 ctp 工作的时序，且不影响整个模块加载的时序要求或者是唤醒时序要求。

5.驱动调试

设备驱动中添加了调试信息，可能根据需要进行相应的信息打印而不需要进行驱动的重新编译导入等工作。驱动调试的关键点以及步骤如下：

5.1. 调试信息的使用方法

设备驱动中添加了模块的参数之后，可以根据需要将想要的打印信息打印出来。需要进行的操作为，对模块的参数进行设置。需要到的工具为 adb shell。仍以加载 ft5x_ts.ko 为例。

步骤如下：

（1）进入 adb shell 界面，使用命令：adb shell

（2）进入模块节点，使用命令：

```
root@android:/ # cd sys/module/ft5x_ts/parameters
```

（3）查看模块参数名称

```
root@android:/sys/module/ft5x_ts/parameters # ls
ls
ctp_debug
```

（4）查看模块参数当前值：

```
root@android:/sys/module/ft5x_ts/parameters # cat ctp_debug
cat ctp_debug
0
```

（5）设置模块参数数值，打印 SUSPEND 信息：

```
root@android:/sys/module/ft5x_ts/parameters # echo 0x02 > ctp_debug
echo 0x02 > ctp_debug
```

（6）查看设置是否成功：

```
root@android:/sys/module/ft5x_ts/parameters # cat ctp_debug
cat ctp_debug
2
```

注意 # 号后面的为命令，然后键入回键即可。echo 命令中，“>”号，前后均为空格。

5.2. 查看驱动是否加载成功

当系统起来之后，触摸无反应，首先检查驱动是否加载成功，probe 函数是否按流程完成。可以通过 adb 工具进行查看，一些简单的 adb 的命令如下所示：

```
adb shell   登录设备的shell
adb push xx.ko /drv     将触摸驱动通过adb工具push到机器中
cd  /system/vendor/modules 进入KO文件目录
ls   *.ko 查看机器中已经有了那些驱动
lsmod   查看系统中已经加载了那些模块
rmmod  ** 卸载驱动 （注：不用加后缀）
insmod  **.ko 加载驱动
getevent  查看系统中已经注册了那些input设备(当触摸有效时，触摸屏幕，会有相应的打印信息)
```

（1）、使用 lsmod 命令查看驱动是否加载

（2）、在 adb shell 中使用 cat /proc/kmsg 命令，或者是使用串口查看内核的打印信息，查看不能正常加载的原因，一般情况下驱动加载不成功的原因有：一是读取的 sys_config1.fex 文件中的配置信息与加载的驱动不匹配，二是 probe 函数遇到某些错误没能正确的完成 probe 的时候返回，三是驱动与所使用的固件不匹配。

5.3. I2C 通信是否成功

如果 detect 函数中使用的为读取设备 chip ID 值，以及检测通信是否成功的方式，则可以根据 detect 函数进行判断 I2C 是否能正常的通信。当 detect 函数中，不使用这两种方法时，可以在 probe 函数中添加调用 ctp_i2c_test 函数进行 I2C 是否能成功通信的判断，或者是在数据读取的时候进行判断。

如果 I2C 通信失败，请先查看 I2C 通信失败的原因。

（1）查看 I2C 的电压是不是正常的，如果 I2C 电压不正常，查看硬件说明地方将 I2C 的电压弄不正常。

（2）确认 I2C 地址是不是正确的，特别是设备可以配置为多个地址的时候。

（3）通过打印信息查看 I2C 通信不成功的原因。

打印信息为：`START Can’t Sendout!` 则有可能为引脚的电压不正确导致通信失败。

打印信息为：`Incomplete xfer <0x20>` 则有可能为 I2C 的地址有问题。

5.4. 是否有中断

在中断处理函数中添加一条打印信息，当触摸屏幕的时候，查看是否打印该信息。如果没有相关的打印信息来，调试的步骤如下：

（1）、驱动中是否进行正确的中断申请，通过打印信息查看中断申请是否已经成功。

（2）、sys_config1.fex 中的中断引脚与驱动中的中断引脚是否匹配。

（3）、主控这边是不是无法接收到中断，将主控与触摸 IC 的中断引脚断开，手动的给主控的中断引脚一个脉冲，查看是否有中断发生。

（4）、查看触摸 IC 是否有中断发生。使用示波器查看，当触摸屏幕的时候，是否有脉冲波形。

5.5. 读取数据分析

将读取到的数据包都打印出来，根据触摸 IC 数据包的格式进行分析，查看数据包的正确性，如果读取的数据包不正确，首先检查驱动中设置的中断触发条件与触摸 IC 设置的中断触发条件是否一致。

6.CTP 使用配置

6.1. sys_config.fex 配置

配置文件 sys_config1.fex 中关于 CTP 的配置项如下：

```
[ctp_para]
ctp_used        = 1
ctp_name        = "gsl2680"
ctp_twi_id      = 1
ctp_twi_addr    = 0x40
ctp_screen_max_x  = 1024
ctp_screen_max_y  = 768
ctp_revert_x_flag  = 0
ctp_revert_y_flag  = 1
ctp_exchange_x_y_flag = 1

ctp_int_port      = port:PA03<6><default><default><default>
ctp_wakeup        = port:PA02<1><default><default><1>
```

文件配置的详细说明可查看第二章中的模块配置介绍的第一节内容。

sys_config.fex 中 ctp 的配置，主要是根据硬件资源，配置 CTP 使用的 i2c 组别号、中断引脚以及 wake up 引脚号、设备地址、CTP 的分辨率等。为了兼容多种不同的 tp 面板，还设置了 ctp_revert_x_flag,ctp_revert_y_flag 以及 ctp_exchange_x_y_flag，这三项的默认值为 0。

ctp_name 的妙用：ctp_name 用于区别不同 tp 面板时需要下载的 tp 参数，目前使用的为 gslX 系列以及 gt82x 系列。

需要反置 x 方向时，若 ctp_revert_x_flag 原值为 0 则将其设置为 1；若 ctp_revert_x_flag 原值为 1 则将其设置为 0。

需要反置 y 方向时，若 ctp_revert_y_flag 原值为 0 则将其设置为 1；若 ctp_revert_y_flag 原值为 1 则将其设置为 0。

需要互换 x 轴跟 y 轴时，若 ctp_exchange_x_y_flag 原值为 0 则将其设置为 1；若 ctp_exchange_x_y_flag 原值为 1 则将其设置为 0。

6.2. 驱动的加载

以模块的形式进行编译，在配置文件中加载触摸驱动。目录：/device/softwinner 然后选择 lunch 下的产品文件夹中的 init.xxx.rc 文件中，加载需要的触摸驱动，如使用 FT5406，那驱动的加载为：

```
#insmod tp_driver
insmod /system/vendor/modules/ft5x_ts.ko
```

若使用自动检测功能时，不需要增加该语句，只需要在该文件中增加：insmod /vendor/modules/sw_device.ko 自动检测驱动，同时 sysconfig.fex 文件中需要增加 xxx_list_para 的配置选项，该配置项的详细信息，请查看第二章。

6.3. IDC 文件

Android4.0 之后，配置文件中需要一个 idc 文件来识别输入设备为触摸屏还是鼠标，如果没有该文件，则默认为鼠标，因此需要添加该文件。

使用 adb shell getevent 命令，当设备的名称为“gslX680”，“gt82x”，“ft5x_ts”，“sunxi-ts”，“gt818_ts”，“gt811”，“gt9xx”，“sw-ts”，“zet622x”时，使用的 idc 名字均为 tp.idc。idc 文件放置的目录为:system/usr/idc ,则在配置文件为 wing_xx.mk 拷贝语句如下所示

```
PRODUCT_COPY_FILES += \
device/softwinner/wing-xxx/sw-keyboard.kl:system/usr/keylayout/sw-keyboard
.kl \
        device/softwinner/wing-xxx/tp.idc:system/usr/idc/tp.idc \
```

当使用 adb shell getevent 命令得到的设备名称与以上的不符合，则需要增加该名称的 idc 文件进行相应的匹配。如使用 getevent 命令后，获得的名称为 ctp_name,如下：

```
root@android:/ # getevent
getevent
add device 1: /dev/input/event3
  name:     "ctp_name"
add device 2: /dev/input/event2
  name:     "bma250"
add device 3: /dev/input/event0
  name:     "sw-keyboard"
could not get driver version for /dev/input/mice, Not a typewriter
add device 4: /dev/input/event1
  name:     "axp20-supplyer"
```

则相应的 idc 文件就应该为 ctp_name.idc ,则在配置文件为 wing_xx.mk 拷贝语句如下所示：

```
#input device config
PRODUCT_COPY_FILES += \
        device/softwinner/wing-xxx/sw-keyboard.kl:system/usr/keylayout/sw-keyboard.kl \
        device/softwinner/wing-xxx/ctp_name.idc:system/usr/idc/ctp_name.idc \
        device/softwinner/wing-xxx/gsensor.cfg:system/usr/gsensor.cfg
```

Idc文件的内容均一致，使用时只需要修改为需要的文件名称即可。

6.4. gslX680 使用说明

gslX680 驱动兼容 gsl1680，gsl2680，gsl3680。为了区分下载的参数，在

sys_config.fex 的配置文件中，需要增加 ctp_name 进行区别，目前，gslX680 系列的参数设置方式为每一种分辨率或者是一组参数设置为一个.h 文件，使用 ctp_name 进行区分，使用时请注意项目中使用的头文件。如使用的参数为“gsl168.h”，则 sysconfig.fex 中的参数如下所示：

```
[ctp_para]
ctp_used                = 1
ctp_twi_id              = 2
ctp_name                = "gsl1680"
ctp_screen_max_x        = 1024
ctp_screen_max_y        = 600
```

当使用的 tp 面板变化时或者使用另外的一组参数，则需要将该参数按照 gsl1680.h 文件的模式进行填写，参数的名称应该易于识别，由于数据比较多，复制时请确保数据完整。使用该组参数时，则需要更换 sysconfig.fex 中 ctp_para 中的 ctp_name 为设置的头文件的参数数组名称。现举例如下所示：

（1）增加头文件

如需要增加一组 1680 的 1280*800 分辨率参数，则命名为 gsl1680_3.h,该头文件中，数组的名称设置为：GSL1680_FW_3

（2）驱动中的增加相关的头文件以及 gslx680_fw_grp 数组中增加相对应的描述信息

该头文件放置在 lichee\linux-3.x\drivers\input\sw_touchscreen 下，需要在 gslX680.c 源文件中添加该头文件以及在支持的 tp 参数数组中增加该参数信息，如下：

```
#include "gslX680.h"
#include "gsl1680.h"          //resolution:1024*600
#include "gsl1680_1.h"        //resolution:800*480
#include "gsl1680_2.h"        //resolution:1024*600
#include "gsl2680.h"          //resolution:1024*768
#include "gsl3680.h"          //resolution:2048*1563
#include "gsl1680_3.h"        //resolution:1024*800
struct gslX680_fw_array {
    const char* name;
    unsigned int size;
    const struct fw_data *fw;
} gslx680_fw_grp[] = {
    {"gsl1680",     ARRAY_SIZE(GSL1680_FW),   GSL1680_FW},
    {"gsl1680_1",   ARRAY_SIZE(GSL1680_FW_1), GSL1680_FW_1},
    {"gsl1680_2",   ARRAY_SIZE(GSL1680_FW_2), GSL1680_FW_2},
```

```
    {"gsl1680_3",    ARRAY_SIZE(GSL1680_FW_3), GSL1680_FW_3},
    {"gsl2680",      ARRAY_SIZE(GSL2680_FW),   GSL2680_FW},
    {"gsl3680",      ARRAY_SIZE(GSL3680_FW),   GSL3680_FW},
};
```

（3）使用时，替换 sysconfig.fex 中的 ctp_name 为当前使用的头文件参数名称
ctp_name 中写入的名称与 gslx680_fw_grp 数组中的 name 值对应
使用此组参数时，sysconfig.fex 中 ctp_para 下的 ctp_name 替换为“gsl1680_3”,如下：

```
[ctp_para]
ctp_used                = 1
ctp_twi_id              = 2
ctp_name                = "gsl1680_3"
ctp_screen_max_x        = 1280
ctp_screen_max_y        = 800
```

6.5. GT（汇顶）系列使用说明

6.5.1 gt 系列驱动介绍

gt 系列的产品在驱动端初始化时需要根据具体的 tp 屏下载相应的参数之后才可以正常的工作，在掉电之后也需要通过驱动端重新下载相关的参数。

gt 系列的驱动包括（gt811，gt82x，gt9xx）为了兼容多款 tp 面板，多种分辨率，目前将 gt 系列的参数抽取出来放置单独的头文件中，通过名字进行匹配的方法进行参数的选择。如果驱动中没有找到相对应的匹配名字，将使用第 0 组参数。

gt82x.ko 驱动为兼容 gt813，gt827，gt828 这三颗 IC 的驱动。gt82x 对应的参数头文件：lichee\linux-3.x\drivers\input\sw_touchscreen\gt82x.h

gt811.ko 驱动为 gt811 这颗 IC 的驱动。头文件中放置了两组参数，gt811 对应的头文件：lichee\linux-3.x\drivers\input\sw_touchscreen\gt811_info.h

gt9xx_ts.ko 驱动为 gt9 系列对应的驱动。头文件中放置了两组 gt911 使用的参数。gt9xx 对应的头文件：lichee\linux-3.x\drivers\input\sw_touchscreen\gt9xx_info.h

6.5.2 增加新 tp 参数的步骤

当新增加一个 tp 的参数，步骤如下：
（1）在头文件中增加参数数组，且数组的名称易于识别。
（2）在驱动中的 xxx_cfg_grp 数组中，添加新参数的名称（头文件中的定义），大小以及数据数组名称等信息。xxx_cfg_grp 中的 xxx 分别为 gt82x，gt9xx，gt811。
（3）在 sysconfig.fex 文件下的 ctp_para 下的 ctp_name 替换为目前使用的 tp 参数的名称，

即 xxx_cfg_grp 中定义的名称。

6.5.3 增加新 tp 参数的例子

以 gt82x 为例子进行相关的说明。

在函数 ctp_get_system_config 中，解析 sysconfig.fex 中的 ctp_name，通过获得的名称查找 gt82x_cfg_grp 中是否有相匹配的名称。goodix_init_panel 函数中通过找到的 index 下载相对应的参数，如果找不到匹配的名称将返回直接返回 0，即下载第 0 组。

增加新参数的步骤如下：

(1) 头文件中增加新的参数

如增加一组 gt813 的 1024 * 600 分辨率的参数时，需要在 gt82x.h 中增加一组参数，命名为：uint8_t gt813_tp1[]；拷贝的时候注意数据的完整性，参数由 tp 屏厂提供，应该为 114 个数据，注意，前两个数据（0x0F、0x80）为寄存器地址，不要变动。

（2）驱动中 xxx_cfg_grp 增加相关信息

在 gt82x.c 中增加相关的信息，支持该组参数，如下所示：

```
struct gt82x_cfg_array {
    const char*         name;
    unsigned int        size;
    uint8_t             *config_info;
} gt82x_cfg_grp[] = {
    {"gt813_evb",    ARRAY_SIZE(gt813_evb),   gt813_evb},
{"gt813_tp1",    ARRAY_SIZE(gt813_tp1),   gt813_tp1},
{"gt828",        ARRAY_SIZE(gt828),       gt828},
};
```

（3）sysconfig.fex 中替换 ctp_name 的内容

使用时，sysconfig.fex 文件中的 ctp_name 应该替换为“gt813_tp1”，如下：

```
[ctp_para]
ctp_used                = 1
ctp_twi_id              = 2
ctp_name                = "gt813_tp1"
ctp_screen_max_x        = 1024
ctp_screen_max_y        = 600
```

6.6 ft5x 系列使用说明

Ft5x 系列兼容 ft5x06 系列以及 ft5x02 系列。

6.6.1 ft5x02 系列使用说明

ft5x02 使用时需要 tp 相关的头文件信息即使用 tp 的 ft5x02_config.h 文件。如果该文件为拷贝过来，则请注意头文件中定义的名称是否与原来的文件一致，特别是文件中定义的变量名称的大小写。

驱动中通过读取 a3 寄存器，通过对其值的判断确定是否为 0x02，如果为 0x02，则说明为 02 系列。此时将通过驱动下载 ft5x02_config.h 相关的参数。当 ic 掉电之后重新上电也需要下载参数。

当发现 tp 无法正常读取数据时，请确认相关的参数是否已经正确的下载。

6.6.2 ft5x06 系列使用说明

ft5x06 系列使用时不需要下载参数。当需要下载固件时，可以打开驱动的 CONFIG_SUPPORT_FTS_CTP_UPG 的定义，通过下载 i 文件去下载固件。正常情况下该定义被屏蔽掉，当确认需要下载时，请打开该宏定义且请更换正确的点 i 文件，否则将造成 tp 无法正常使用的情况。

7. CTP 测试

TP 的使用中，主要关注点以及影响用户体验点均为性能与稳定性，性能主要关注线性度，精度，响应速度，识别点数（且要求多指识别无交叉）。可靠性关注充电是否有干扰，在模式切换时是否出现失效，如有水，进入待机，长时间开机等。

如何简单的判断一个 CTP 的好坏，给出一些测试的参考条目以及简单的分析结果。

7.1.性能测试

使用工具：Dev Tools 下的 Pointer Location 按照顺序进行测试的条目：

（1）是否能画点，单点，多点，多点时是否有交叉。

（2）坐标值是否正确，确定原点是否正确，沿 X，Y 划线，查看坐标值是否连续。Pointer Location 的上方显示按下的点数值，坐标值，按点大小等数值。

（3）是否能画直线，划线是否流畅，即画出的线条无明显的阶梯状，毛刺等。

（4）单指或者多指以非常慢的速度画线，看触摸点是否能按照手指的方向走，是否有断线，多指时线条是否有交叉等现象。

（5）单指或者多指以非常快的速度画线，看触摸点是否能快速按照手指的方向走，是否有断线，多指划线时线条是否有交叉等现象。

（6）单指或者多指画对角线，看线性度如何，即画出的线条是否流畅，是否会出现断线，阶梯状，毛刺等。

（7）单指或者多指是否能画圆，方等各种多边形图案。

（8）两个手指由比较远的距离即可以画出两条线，慢慢靠近，查看多近的时候，不能画出两条直线来简单的判断 tp 的精度。

7.2. 可靠性测试

（1）正常操作 TP，是否出现异常。

（2）使用机器配置的电源或者是插 USB 线，操作 TP，查看是否会出现异常声音。

（3）多次休眠唤醒，查看 TP 是否能正常的进入休眠以及唤醒。查看进入休眠是否成功的简单方法为使用 adb shell 的 getevent 命令，当进入休眠时，查看是否有上报点值。

（4）将手指弄湿，操作 TP，查看是否能正常操作。

（5）使用切西瓜游戏查看 TP 的可靠性，即长时间的操作 TP，查看是否出现失效等现象。

7.3. 现象分析

（1）坐标已经上报，但是不能画点时，主要的原因是驱动中收到了按下的事件，但是没有上报手指抬起时的事件。

（2）坐标值不正确时，确认是否 x，y 的坐标值反了，x，y 反了，可以通过配置文件进行修改，偏离原来的坐标点或者是坐标不连续时，查看驱动中数据包的解析是否正确以及触摸 IC 的一些相关设置等

（3）画点，画线，画各种多边形图案，可以简单的判断 CTP 的性能，可能出现下面的现象：

1）正常操作 TP 时，是否出现卡顿，体验差，无法正常的画出想要的各种多边形。

2）手指画过的地方没有出现线条。

3）画的线条不流畅即画出的线条出现阶梯状，毛刺等情况。

4)手指没有抬起但是出现断线。

5)乱报点即在偏离手指划线的方向出现毛刺尖角的形状。
6)线性度不佳即画对角线时出现明显的阶梯状。

出现这些现象更多的是结合触摸驱动以及触摸 IC 的固有参数去调试。

驱动中需要关注的地方：

1)驱动中是否打印太多的信息导致体验差，卡顿等。
2)驱动中断的触发条方式是否与触摸 IC 固件中的触发方式一致。
3)中断处理函数中打印信息是否太多，导致错过了读数据的时间，从而漏读数据。

根据屏的大小，结构以及材料等不同将导致触摸 IC 的灵敏度，按点阈值等固有参数不一样，触摸效果差，需要根据 TP 屏的特点对这些参数进行调整。

(4)CTP 的可靠性测试中，可能出现的现象有：

1)操作 TP 或者是插入充电器，USB 线等出现杂音。插入充电器出现杂音时，查看是 否触摸点的上报率与电源的频率点有冲突造成。操作 TP 时出现杂音，是否 CTP 的上报速度与 I2C 的速度有影响，可以通过调整 I2C 的速度或者是调整 CTP 的频率。

2)CTP 不能正常的进入休眠，或者是休眠之后无法唤醒。出现该问题时，主要查看 datashe 中给出的 CTP 进入休眠与唤醒的条件，查看进入时的打印信息，是否已经操作成功。

3)当手指湿的时候操作 TP，出现乱报点，失效等现象。出现该现象与 tp 的结构以及材料有关系，可以查看相关参数是否正确设置。

4)长时间操作 TP，出现失效，不灵敏等现象。出现失效的时候可以通过 adb shell 的 getevent 命令，查看失效时候是否有数据上报，没有数据上报则问题出现在 CTP 端。如果上报数值时，没有响应，在问题在主控端。

8.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.