本文描述在RK3126平台上添加一个新的TP驱动（gslx680驱动）以及详细的驱动代码信息。如有不足之处，敬请指出。

1、修改dts，添加新的i2c设备。

在 arch/arm/boot/dts/rk312x-sdk-v2.2.dtsi中添加i2c设备的相关信息：

ts@40 {

compatible = "gslX680";

reg = <0x40>;

wake-gpio = <&gpio0 GPIO\_D3 GPIO\_ACTIVE\_LOW>;

irp-gpio = <&gpio0 GPIO\_A2 IRQ\_TYPE\_LEVEL\_HIGH>;

revert\_x = <0>;

revert\_y = <0>;

};

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

&i2c2 {

status = "okay";

/\*

ts@55 {

compatible = "goodix,gt8xx";

reg = <0x55>;

touch-gpio = <&gpio1 GPIO\_B0 IRQ\_TYPE\_LEVEL\_LOW>;

reset-gpio = <&gpio2 GPIO\_C1 GPIO\_ACTIVE\_LOW>;

//power-gpio = <&gpio0 GPIO\_C5 GPIO\_ACTIVE\_LOW>;

max-x = <1280>;

max-y = <800>;

};\*/

ts@40 {

compatible = "gslX680";

reg = <0x40>;

//wake-gpio = <&gpio0 GPIO\_D3 GPIO\_ACTIVE\_LOW>;

irp-gpio = <&gpio0 GPIO\_A2 IRQ\_TYPE\_LEVEL\_HIGH>;

revert\_x = <0>;

revert\_y = <0>;

};

/\* ... \*/

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

表示i2c2总线上下挂在了多个i2c设备。   
其中ts@40是表示此i2c设备的设备类型为触摸屏，设备地址为0x40(7位地址，注意：在i2c的传输函数中，会将此地址左移一位，因此实际上gslx680的i2c设备地址为0x80)。该节点下有多个属性：   
1、compatible = "gslX680";属性用于驱动和设备的绑定。表示特定的设备名称，此处为gslX680；   
2、reg = <0x40>;属性表示此设备的i2c地址为0x40，等同于@40；   
3、wake-gpio = <&gpio0 GPIO\_D3 GPIO\_ACTIVE\_LOW>;表示复位引脚使用的是GPIO0 中的GPIO\_D3这个引脚，低电平有效。   
irp-gpio = <&gpio0 GPIO\_A2 IRQ\_TYPE\_LEVEL\_HIGH>;表示中断引脚使用的是GPIO0中的GPIO\_A2这个引脚，高电平触发。   
很奇怪，为什么这里没有上电的信息，以及在整个驱动程序中都没有给ic上电的操作。在前面的MTK平台上的tp驱动都有上电的动作，暂时还搞不懂在RK平台上为什么没有。   
4、revert\_x = <0>; revert\_y = <0>;标记x和y是否需要翻转。   
在上述的信息中，可以通过of接口获取到属性对应的值。在后面的probe()函数中就会使用到。

注：关于dts的详细信息可以查看[ARM Linux 3.x的设备树（Device Tree）](http://blog.csdn.net/21cnbao/article/details/8457546)和[Device Tree Usage](http://elinux.org/Device_Tree_Usage#Initial_structure)

2、修改Makefile、Kconfig、defconfig

(1)、修改Makefile添加gslx680驱动

在 drivers/input/touchscreen/Makefile中添加驱动：   
obj-$(CONFIG\_TOUCHSCREEN\_GSLX680) += gslx680/。   
只要当配置了CONFIG\_TOUCHSCREEN\_GSLX680的选项才会去编译gslx680目录下的内容。在配置内核的时候会通过make menuconfig来配置对应的选项。或者是直接在defconfig文件中强制设置该选项。

注：如果不想要这么复杂，可以将该语句写成obj-y += gslx680/来强制编译该驱动。

(2)、修改Kconfig添加驱动配置描述

在 drivers/input/touchscreen/Kconfig中添加驱动配置描述：

config TOUCHSCREEN\_GSLX680

tristate "gslX680 touchscreen driver"

help

gslX680 touchscreen driver

* 1
* 2
* 3
* 4

(3)、配置defconfig设置编译驱动

一般在内核中会有配置好的默认的config文件供参考，可以直接修改defconfig来选择编译某个驱动。此处在arch/arm/configs/rockchip\_defconfig文件中添加CONFIG\_TOUCHSCREEN\_GSLX680=y并将该文件拷贝到kernel目录下命名为.config即可。

2、添加i2c驱动

#define GSLX680\_I2C\_NAME "gslX680"

#define GSLX680\_I2C\_ADDR 0x40

static const struct i2c\_device\_id gsl\_ts\_id[] = {

{GSLX680\_I2C\_NAME, 0},

{}

};

MODULE\_DEVICE\_TABLE(i2c, gsl\_ts\_id);

static struct i2c\_driver gsl\_ts\_driver = {

.driver = {

.name = GSLX680\_I2C\_NAME,

.owner = THIS\_MODULE,

},

#ifndef CONFIG\_HAS\_EARLYSUSPEND

// .suspend = gsl\_ts\_suspend,

// .resume = gsl\_ts\_resume,

#endif

.probe = gsl\_ts\_probe,

.remove = gsl\_ts\_remove,

.id\_table = gsl\_ts\_id,

};

static int \_\_init gsl\_ts\_init(void)

{

int ret;

printk("==gsl\_ts\_init==\n");

ret = i2c\_add\_driver(&gsl\_ts\_driver);

printk("ret=%d\n",ret);

return ret;

}

static void \_\_exit gsl\_ts\_exit(void)

{

printk("==gsl\_ts\_exit==\n");

i2c\_del\_driver(&gsl\_ts\_driver);

return;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38

注册名字为GSLX680\_I2C\_NAME的i2c驱动，即gslx680，该驱动支持的设备名为字gsl\_ts\_id[]里的设备名称。因为我们在dts中已注册了一个名字为gslx680的i2c设备。因此，设备与驱动可以匹配成功并正确执行probe()函数。   
至于设备与驱动是如何匹配的，可以参照[Linux i2c子系统](http://blog.csdn.net/encourage2011/article/details/51723154)

3、gsl\_ts\_probe()

static int gsl\_ts\_probe(struct i2c\_client \*client,const struct i2c\_device\_id \*id)

{

struct gsl\_ts \*ts;

int rc;

struct device\_node \*np = client->dev.of\_node;

enum of\_gpio\_flags wake\_flags;

unsigned long irq\_flags;

// 检查i2c适配器的能力

printk("GSLX680 Enter %s\n", \_\_func\_\_);

if (!i2c\_check\_functionality(client->adapter, I2C\_FUNC\_I2C))

{

dev\_err(&client->dev, "I2C functionality not supported\n");

return -ENODEV;

}

// 为ts申请内核空间

ts = kzalloc(sizeof(\*ts), GFP\_KERNEL);

if(!ts)

return -ENOMEM;

printk("==kzalloc success=\n");

ts->client = client;

i2c\_set\_clientdata(client, ts);

ts->device\_id = id->driver\_data;

// 从设备节点np中获取到irq和wake 的gpio的信息

ts->irq\_pin=of\_get\_named\_gpio\_flags(np, "irp-gpio", 0, (enum of\_gpio\_flags \*)&irq\_flags);

ts->wake\_pin=of\_get\_named\_gpio\_flags(np, "wake-gpio", 0, &wake\_flags);

// 为设备申请gpio，并设置默认电平

if(gpio\_is\_valid(ts->wake\_pin))

{

rc = devm\_gpio\_request\_one(&client->dev, ts->wake\_pin, (wake\_flags & OF\_GPIO\_ACTIVE\_LOW) ? GPIOF\_OUT\_INIT\_LOW : GPIOF\_OUT\_INIT\_HIGH, "gslX680 wake pin");

if(rc != 0)

{

dev\_err(&client->dev, "gslX680 wake pin error\n");

return -EIO;

}

g\_wake\_pin = ts->wake\_pin;

//msleep(100);

}

else

{

dev\_info(&client->dev, "wake pin invalid\n");

}

if(gpio\_is\_valid(ts->irq\_pin))

{

rc = devm\_gpio\_request\_one(&client->dev, ts->irq\_pin, (irq\_flags & OF\_GPIO\_ACTIVE\_LOW) ? GPIOF\_OUT\_INIT\_LOW : GPIOF\_OUT\_INIT\_HIGH, "gslX680 irq pin");

if (rc != 0)

{

dev\_err(&client->dev, "gslX680 irq pin error\n");

return -EIO;

}

}

else

{

dev\_info(&client->dev, "irq pin invalid\n");

}

// 创建工作队列，申请input设备

rc = gslX680\_ts\_init(client, ts);

if(rc < 0)

{

dev\_err(&client->dev, "GSLX680 init failed\n");

goto error\_mutex\_destroy;

}

gsl\_client = client;

// 从设备节点中获取属性信息

of\_property\_read\_u32(np,"revert\_x",&revert\_x);//sss

of\_property\_read\_u32(np,"revert\_y",&revert\_y);//sss

// 初始化IC，包括复位，测试i2c以及加载ic配置信息

init\_chip(ts->client);

check\_mem\_data(ts->client);

// 申请中断号

ts->irq=gpio\_to\_irq(ts->irq\_pin); //If not defined in client

if (ts->irq)

{

// 为client->dev设备的中断号ts->irq申请irq\_flags触发的中断，中断服务子程序为gsl\_ts\_irq

rc = devm\_request\_threaded\_irq(&client->dev, ts->irq, NULL, gsl\_ts\_irq, irq\_flags | IRQF\_ONESHOT, client->name, ts);

if(rc != 0)

{

printk(KERN\_ALERT "Cannot allocate ts INT!ERRNO:%d\n", rc);

goto error\_req\_irq\_fail;

}

//disable\_irq(ts->irq);

}

else

{

printk("gsl x680 irq req fail\n");

goto error\_req\_irq\_fail;

}

ts->tp.tp\_resume = gsl\_ts\_late\_resume;

ts->tp.tp\_suspend = gsl\_ts\_early\_suspend;

tp\_register\_fb(&ts->tp);

#ifdef CONFIG\_HAS\_EARLYSUSPEND

ts->early\_suspend.level = EARLY\_SUSPEND\_LEVEL\_BLANK\_SCREEN + 1;

//ts->early\_suspend.level = EARLY\_SUSPEND\_LEVEL\_DISABLE\_FB + 1;

ts->early\_suspend.suspend = gsl\_ts\_early\_suspend;

ts->early\_suspend.resume = gsl\_ts\_late\_resume;

register\_early\_suspend(&ts->early\_suspend);

#endif

#ifdef GSL\_MONITOR

printk( "gsl\_ts\_probe () : queue gsl\_monitor\_workqueue\n");

INIT\_DELAYED\_WORK(&gsl\_monitor\_work, gsl\_monitor\_worker);

gsl\_monitor\_workqueue = create\_singlethread\_workqueue("gsl\_monitor\_workqueue");

queue\_delayed\_work(gsl\_monitor\_workqueue, &gsl\_monitor\_work, 1000);

#endif

printk("[GSLX680] End %s\n", \_\_func\_\_);

return 0;

//exit\_set\_irq\_mode:

error\_req\_irq\_fail:

free\_irq(ts->irq, ts);

error\_mutex\_destroy:

input\_free\_device(ts->input);

kfree(ts);

return rc;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131

(1)、自定义的数据结构gsl\_ts

一般在自己的驱动程序中，都会为该驱动程序封装一个数据结构，这里的gsl\_ts就充当这种角色。   
在该驱动程序中自定义了一个数据结构:

struct gsl\_ts {

struct i2c\_client \*client;

struct input\_dev \*input;

struct work\_struct work;

struct workqueue\_struct \*wq;

struct gsl\_ts\_data \*dd;

u8 \*touch\_data;

u8 device\_id;

int irq;

int irq\_pin;

int wake\_pin;

struct tp\_device tp;

#if defined(CONFIG\_HAS\_EARLYSUSPEND)

struct early\_suspend early\_suspend;

#endif

};

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

client表示一个i2c的设备；   
input表示一个输入设备；   
work表示一个工作，用于处理中断到来之后获取坐标等信息；   
wq表示一个工作队列，将上面的work加入到该工作队列中；   
dd和touch\_data用来存储坐标的相关信息；   
device\_id表示i2c设备的设备号；   
irq申请的中断号；   
irq\_pin中断引脚；   
wake\_pin复位引脚；这两个引脚信息可以通过dts获取到.   
tp创建一个为tp\_device的数据结构，里面有个成员notifier\_block用来接收LCD背光灯的亮暗的通知进而调用suspend()和resume()。主要的实现在tp\_suspend.h中。至于这里面的详细机制还不是很明白。

(2)、检查i2c适配器的能力

在很多i2c设备驱动程序中，一进入probe()就要检查i2c适配器的能力。现在还不清楚这么做的目的是什么。后面会仔细的学习一下linux 的i2c子系统。

(3)、获取wake和irq的引脚信息

在前面配置dts说过：dts中设备节点的属性的值可以通过of\_接口获取到。

// 从设备节点np中获取到irq和wake 的gpio的信息

ts->irq\_pin=of\_get\_named\_gpio\_flags(np, "irp-gpio", 0, (enum of\_gpio\_flags \*)&irq\_flags);

ts->wake\_pin=of\_get\_named\_gpio\_flags(np, "wake-gpio", 0, &wake\_flags);

* 1
* 2
* 3

可以通过设备节点np获取到它irq-gpio这一个属性的值存放在ts->irq\_pin中，以及将其flag刚到变量irq\_flags中。因此我们可以知道：

ts->irq\_pin = GPIO\_A2

irq\_flag = IRQ\_TYPE\_LEVEL\_HIGH

* 1
* 2

同理，wake\_pin也是一样。

(4)、申请gpio并设置输出电平

将irq和wake 引脚电平都设置输出低电平。

(5)、gslX680\_ts\_init

static int gslX680\_ts\_init(struct i2c\_client \*client, struct gsl\_ts \*ts)

{

struct input\_dev \*input\_device;

int rc = 0;

printk("[GSLX680] Enter %s\n", \_\_func\_\_);

// 配置获取坐标信息

ts->dd = &devices[ts->device\_id];

if(ts->device\_id == 0)

{

ts->dd->data\_size = MAX\_FINGERS \* ts->dd->touch\_bytes + ts->dd->touch\_meta\_data;

ts->dd->touch\_index = 0;

}

// 申请空间存放坐标信息

ts->touch\_data = kzalloc(ts->dd->data\_size, GFP\_KERNEL);

if(!ts->touch\_data)

{

pr\_err("%s: Unable to allocate memory\n", \_\_func\_\_);

return -ENOMEM;

}

// 申请一个input\_dev 设备

input\_device = input\_allocate\_device();

if (!input\_device) {

rc = -ENOMEM;

goto error\_alloc\_dev;

}

// 初始化input\_device

ts->input = input\_device;

input\_device->name = GSLX680\_I2C\_NAME;

input\_device->id.bustype = BUS\_I2C;

input\_device->dev.parent = &client->dev;

input\_set\_drvdata(input\_device, ts);

//

#ifdef REPORT\_DATA\_ANDROID\_4\_0

\_\_set\_bit(EV\_ABS, input\_device->evbit);

\_\_set\_bit(EV\_KEY, input\_device->evbit);

\_\_set\_bit(EV\_REP, input\_device->evbit);

\_\_set\_bit(INPUT\_PROP\_DIRECT, input\_device->propbit);

input\_mt\_init\_slots(input\_device, (MAX\_CONTACTS+1),0);

#else

input\_set\_abs\_params(input\_device,ABS\_MT\_TRACKING\_ID, 0, (MAX\_CONTACTS+1), 0, 0);

set\_bit(EV\_ABS, input\_device->evbit);

set\_bit(EV\_KEY, input\_device->evbit);

\_\_set\_bit(INPUT\_PROP\_DIRECT, input\_device->propbit);

input\_device->keybit[BIT\_WORD(BTN\_TOUCH)] = BIT\_MASK(BTN\_TOUCH);

#endif

#ifdef HAVE\_TOUCH\_KEY

input\_device->evbit[0] = BIT\_MASK(EV\_KEY);

//input\_device->evbit[0] = BIT\_MASK(EV\_SYN) | BIT\_MASK(EV\_KEY) | BIT\_MASK(EV\_ABS);

for (i = 0; i < MAX\_KEY\_NUM; i++)

set\_bit(key\_array[i], input\_device->keybit);

#endif

set\_bit(ABS\_MT\_POSITION\_X, input\_device->absbit);

set\_bit(ABS\_MT\_POSITION\_Y, input\_device->absbit);

set\_bit(ABS\_MT\_TOUCH\_MAJOR, input\_device->absbit);

set\_bit(ABS\_MT\_WIDTH\_MAJOR, input\_device->absbit);

input\_set\_abs\_params(input\_device,ABS\_MT\_POSITION\_X, 0, SCREEN\_MAX\_X, 0, 0);

input\_set\_abs\_params(input\_device,ABS\_MT\_POSITION\_Y, 0, SCREEN\_MAX\_Y, 0, 0);

input\_set\_abs\_params(input\_device,ABS\_MT\_TOUCH\_MAJOR, 0, PRESS\_MAX, 0, 0);

input\_set\_abs\_params(input\_device,ABS\_MT\_WIDTH\_MAJOR, 0, 200, 0, 0);

// 创建工作队列

ts->wq = create\_singlethread\_workqueue("kworkqueue\_ts");

if(!ts->wq)

{

dev\_err(&client->dev, "Could not create workqueue\n");

goto error\_wq\_create;

}

flush\_workqueue(ts->wq);

// 初始化工作 ts->work，其操作为 gslX680\_ts\_worker()

INIT\_WORK(&ts->work, gslX680\_ts\_worker);

// 向input子系统注册一个input\_dev

rc = input\_register\_device(input\_device);

if (rc)

goto error\_unreg\_device;

return 0;

error\_unreg\_device:

destroy\_workqueue(ts->wq);

error\_wq\_create:

input\_free\_device(input\_device);

error\_alloc\_dev:

kfree(ts->touch\_data);

return rc;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96

在gslX680\_ts\_init()中主要做了如下工作：

**配置获取坐标信息**

每次当中断来了之后，就要求通过i2c去读取坐标的信息，至于从哪里读取以及读取多少个，都是通过ts->dd来决定的。

**为存储坐标信息申请空间**

坐标信息放在ts->touch\_data中。

**申请及初始化input\_dev设备，向input子系统注册该设备**

这里面的内容涉及到input子系统，我还没有做过深入的了解。

**初始化工作 ts->work**

ts->work对应的操作为gslX680\_ts\_worker()，在中断来了之后，会queue\_work(ts->wq, &ts->work);让ts->work工作起来，就会去读取坐标等信息，然后通过input子系统上报给Android系统。

(6)、获取属性信息

通过of\_接口获取revert\_x和revert\_y的信息，以此来决定坐标是否要翻转。

(7)、初始化ic

初始化的内容会放到一个全局的数组之中，这项工作一般都要FAE来完成。

(8)、申请中断号以及中断服务子程序

通过devm\_request\_threaded\_irq接口为设备申请一个中断服务子程序gsl\_ts\_irq()，触发方式为irq\_flags即IRQ\_TYPE\_LEVEL\_HIGH高电平触发。

(9)、配置休眠唤醒

在前面说过，tp的休眠唤醒是通过LCD亮暗屏来决定的，这个动作由tp\_register\_fb()来实现。

ts->tp.tp\_resume = gsl\_ts\_late\_resume;

ts->tp.tp\_suspend = gsl\_ts\_early\_suspend;

tp\_register\_fb(&ts->tp);

* 1
* 2
* 3

注：如果申请资源出错的话一定要记得释放资源以及前面的资源。比如说这里为ts申请的内核空间、申请的中断号、申请的input设备、申请的工作队列。

上述probe()配置完成之后就是等待中断，如果中断到来，关闭中断，启动工作去读取坐标等信息并通过input子系统上报，之后再使能中断。如此反复。

4、中断服务子程序

static irqreturn\_t gsl\_ts\_irq(int irq, void \*dev\_id)

{

struct gsl\_ts \*ts = dev\_id;

print\_info("========gslX680 www.baiyuewang.net Interrupt=========\n");

disable\_irq\_nosync(ts->irq);

if (!work\_pending(&ts->work))

{

queue\_work(ts->wq, &ts->work);

}

return IRQ\_HANDLED;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

一旦有中断到来，立马调用gsl\_ts\_irq()，在这个中断服务子程序中先判断ts->work是否挂起，如果没有挂起就启动工作队列ts->wq的工作ts->work。ts->work与gslX680\_ts\_worker()对应，主要用来读取坐标信息。

5、休眠唤醒

关于休眠和唤醒的内容根据ic的特性设置。如休眠的时候需要关闭中断、配置进入休眠模式、拉低wake引脚。唤醒的时候唤醒ic，使能wake引脚、使能中断等。