Android Input

开发指南

版本号: 2.1
发布日期: 2023.07.21

版本历史

版本号	日期	制/修订人	内容描述
1.0	2020.11.11	AW	初始版本文档。
1.1	2020.12.31	AW	更新部分配置路径。
2.0	2023.03.09	AW1696	1. 增加 aw input 框架介绍。 2. 完善 TP 模组适配过程。 3. 增加调试章节，介绍常用命令。
2.1	2023.07.21	AWA1979	添加 nvt36xxx spi tp 支持

目　录

1 前言

1.1 编写目的

为达到能快速使用的目的。文档对常用外设的加载原理，使用方法步骤，如何添加一个新的模组等做了详细的讲解。

1.2 适用范围

介绍 AW input 模块设计，适用于 AW Android 平台。

1.3 相关人员

相关的开发与维护人员应该仔细阅读本文档。

2 AW Input 框架介绍

为了便于快速适配新的驱动，AW 设计了 input 框架，方便快速适配一款新外设。

当前内核 input 基本框架如下。

```
driver/input/
├── ctp              //存放TP驱动，部分版本命名为touchscreen
│   ├── cyttsp5
│   ├── gslx680new       //gsl驱动
│   ├── gt9xxnew
│   ├── Kconfig
│   ├── Makefile
│   └── wacom_pen
├── init-input.c         //全志input框架核心
├── init-input.h         //全志input框架h文件
├── Kconfig
├── Makefile
├── misc             //存放没有具体分类的input驱动，如马达，霍尔等
│   ├── hall_sensor      //霍尔传感器
│   ├── Kconfig
│   ├── Makefile
│   └── vibrator         //马达
└── sensor           //存放sensor，主要为gsensor,lightsensor等
    ├── Kconfig
    ├── Makefile
    ├── mir3da
    ├── msa
    ├── sc7a20
    └── stk3x1x
```

下面主要介绍全志 input 核心框架。

2.1 init-input 介绍

2.1.1 input 设备类型区分

当前 input 设备分类主要如下。

定义在 init-input.h。

```
enum input_sensor_type {
    CTP_TYPE,           //CTP类型
    GSENSOR_TYPE,       //gsensor类型
    GYR_TYPE,           //陀螺仪类型
    COMPASS_TYPE,
    LS_TYPE,            //light sensor类型
```

```
    MOTOR_TYPE,
};
```

其中最常用是 gsensor 和 ctp 类型，其他类型使用较少。

2.2 设备结构体

2.2.1 sensor 类型

sensor 类型结构体如下。

```
struct sensor_config_info {
    enum input_sensor_type input_type;    //设备类型，例如GSENSOR_TYPE，LS_TYPE
    int sensor_used;                //用于判断设备是否能使用上
    int isI2CClient;
    __u32 twi_id;                   //配置twi，例如twi0则为0，twi1则为1
    const char *np_name;                //在dts中需要寻找的节点名字，如不配置则GSENSOR_TYPE为gsensor，LS_TYPE为
      light_sensor
    u32 int_number;                 //中断引脚
    struct gpio_config irq_gpio;
    char *ldo;                      //配置电压
    struct device *dev;
    struct pinctrl *pinctrl;
    const char *sensor_power;
    u32 sensor_power_vol;
    struct device_node *node;           //dts里的设备节点
    struct regulator *sensor_power_ldo;
};
```

2.2.2 ctp 类型

ctp 类型结构体如下。

```
struct ctp_config_info {
    enum input_sensor_type input_type;    //设备类型
    int ctp_used;
    int isI2CClient;
    __u32 twi_id;
    const char *name;               //dts里配置的name，用于需要加载fw的驱动，如gsl
    const char *np_name;
    int screen_max_x;               //x轴的最大值
    int screen_max_y;               //y轴的最大值
    int revert_x_flag;              //是否翻转x
    int revert_y_flag;
    int exchange_x_y_flag;              //是否交换xy的值
    int ctp_gesture_wakeup;             //是否支持手势唤醒
    u32 int_number;
    unsigned char device_detect;
    const char *ctp_power;
    u32 ctp_power_vol;
    struct gpio_config ctp_power_io;
    struct regulator *ctp_power_ldo;
```

```
    struct gpio_config irq_gpio;
    struct gpio_config wakeup_gpio;
#ifdef TOUCH_KEY_LIGHT_SUPPORT
    struct gpio_config key_light_gpio;
#endif
    struct device *dev;
    struct device_node *node;
    struct pinctrl *pinctrl;
    int probed;                 //用于判断probe是否成功
};
```

2.2.3 motor 类型

motor 类型结构如下。

```
struct motor_config_info {
    enum input_sensor_type input_type;
    int motor_used;
    int vibe_off;               //motor是否开启
    u32 ldo_voltage;            //启动电压
    const char *ldo;
    struct regulator *motor_power_ldo;
    struct device *dev;
    struct gpio_config motor_gpio;
};
```

2.3 接口设计

aw input 框架的接口设计。

2.3.1 input_request_int

- 函数原型：int input_request_int(enum input_sensor_type *input_type, irq_handler_t handle, unsigned long trig_type, void *para)。
- 作用：用于给特定的设备类型申请中断。
- 参数：
  - input_type：设备类型。
  - handle：中断处理函数。
  - trig_type：中断触发类型。
  - para：传递给 handle 的参数。
- 返回值：
  - 0：sucess。
  - 其他：fail。

2.3.2 input_free_int

- 函数原型：int input_free_int(enum input_sensor_type *input_type, void *para)。
- 作用：释放申请的中断。
- 参数：
  - input_type：设备类型
  - para：传递给 handle 的参数
- 返回值：
  - 0 : sucess
  - 其他：fail

2.3.3 input_set_int_enable

- 函数原型：int input_set_int_enable(enum input_sensor_type *input_type, u32 enable)。
- 作用：使能中断。
- 参数：
  - input_type：设备类型。
  - enable：使能中断，0 为 disable，1 为 enable。
- 返回值：
  - 0 : sucess。
  - 其他：fail。

2.3.4 input_set_int_enable_force

- 函数原型：int input_set_int_enable_force(enum input_sensor_type *input_type, u32 enable)。
- 作用：强制使能中断到 0 或者为 1。
- 参数：
  - input_type：设备类型。
  - enable：使能中断，0 为 disable，1 为 enable。
- 返回值：
  - 0 : sucess。
  - 其他：fail。

2.3.5 input_set_power_enable

- 函数原型：int input_set_power_enable(enum input_sensor_type *input_type, u32 enable)。
- 作用：拉高电平或拉低电平。
- 参数：
  - input_type：设备类型。
  - enable：拉高或拉低，0 为拉低，1 为拉高。
- 返回值：
  - 0 : sucess。
  - 其他：fail。

2.3.6 input_sensor_startup

- 函数原型：int input_sensor_startup(enum input_sensor_type *input_type)。
- 作用：获取 dts 下的驱动配置信息，通常在驱动初始化过程使用。
- 参数：
  - input_type：设备类型。
- 返回值：
  - 0 : sucess。
  - 其他：fail。

2.3.7 input_sensor_init

- 函数原型：int input_sensor_startup(enum input_sensor_type *input_type)。
- 作用：初始化引脚电平信息，通常在 input_sensor_startup 后使用。
- 参数：
  - input_type：设备类型。
- 返回值：
  - 0 : sucess。
  - 其他：fail。

2.3.8 input_sensor_free

- 函数原型：int input_sensor_startup(enum input_sensor_type *input_type)。
- 作用：释放相关资源，通常在驱动卸载时使用。
- 参数：
  - input_type：设备类型。
- 返回值：
  - 0 : sucess。
  - 其他：fail。

3 驱动模块配置信息

本章节主要讲述当前 input 相关外设的 dts 配置信息。

配置文件在如下路径。

```
longan/device/config/chips/{board}/configs/xxx/board.dts
```

说明

linux 4.9 及以后版本已将部分 sensor 和 TP 的相关配置从 sys_config.fex 搬到 board.dts 中，相关模块负责人需留意。

3.1 CTP 通用配置

ctp 一般直接配置到对应的 twi 引脚下，以 gslX680 驱动为例，ctp 的常用配置如下。

```
&twi0 {
    clock-frequency = <400000>;
    pinctrl-0 = <&twi0_pins_a>;
    pinctrl-1 = <&twi0_pins_b>;
    twi-supply = <&reg_cldo3>;
    status = "okay";
    ctp: ctp@0 {
        compatible = "allwinner,gslX680";          // 对应驱动中的 .compatible
        reg = <0x40>;                              // 标示CTP的i2c从机地址，必填
        device_type = "ctp";                       // 标示设备类型为“ctp”
        status = "okay";                           // 标示使能CTP
        ctp_name = "gsl3676_A101E39A02";           // 标示CTP的fw信息，目前gslX680，gslX680new和gt9xx的驱动需要用到此项
        ctp_twi_id = <0x0>;                        // 标示CTP的twi信息
        ctp_twi_addr = <0x40>;                     // 标示CTP的i2c从机地址
        ctp_screen_max_x = <0x500>;                // 标示CTP的x轴范围，16进制
        ctp_screen_max_y = <0x320>;                // 标示CTP的y轴范围，16进制
        ctp_revert_x_flag = <0>;                   // 标示CTP的x轴是否翻转
        ctp_revert_y_flag = <1>;                   // 标示CTP的y轴是否翻转
        ctp_exchange_x_y_flag = <0x1>;             // 标示CTP的x轴、y轴互换
        ctp_int_port = <&pio PH 9 GPIO_ACTIVE_LOW>;  // 标示CTP的中断口配置
        ctp_power_io = <&pio PH 14 GPIO_ACTIVE_LOW>;  // 标示CTP的电源IO控制脚
        ctp_wakeup = <&pio PH 10 GPIO_ACTIVE_LOW>;   // 标示CTP的唤醒口配置
        ctp-supply = <&reg_cldo2>;                 // 标示CTP使用哪路电源
        ctp_power_ldo_vol = <3300>;                // 标示CTP的供电电压，单位mV
    };
};
```

3.2 keyboard 通用配置

目前平台的 keyboard 基本是上使用 lradc 实现的，其配置信息如下。

```
&keyboard {
    compatible = "allwinner,keyboard_1350mv";          // 对应驱动中的 .compatible，其中1350mV为lradc电压值
    status = "okay";                                   // 标示使能keyboard
    key_cnt = <3>;                                     // 标示按键的数量
    key0 = <475 0x7372>;                               // 标示为音量+-两个按键同时按下上报的值，475为电压，0x7372为key值
    key1 = <646 0x73>;                                 // 标示为key1的采样值，646为电压，0x73为key值
    key2 = <900 0x72>;
};
```

说明

keyboard 的配置不包含电源键，reset 键和 uboot 按键。

3.3 Sensor 通用配置

sensor 主要分为 gsensor、lightsensor 和 hallsensor，其几种通用配置如下。

3.3.1 gsensor 配置

```
gsensor {
    compatible = "allwinner,sc7a20";                   // 对应驱动中的 .compatible
    reg = <0x18>;                                      // 标示gsensor的i2c从机地址，必填
    device_type = "gsensor";                           // 标示设备类型为gsensor
    status = "okay";                                   // 标示使能CTP
    gsensor_twi_id = <0x1>;
    gsensor_twi_addr = <0x18>;
    gsensor_int1 = <&pio PH 11 (GPIO_ACTIVE_LOW | GPIO_PULL_UP)>;  // 标示CTP的中断口配置
    gsensor-supply = <&reg_cldo3>;                     // 标示CTP使用哪路电源
    gsensor_vcc_io_val = <3300>;                       // 标示CTP的供电电压，单位mV
};
```

3.3.2 lightsensor 配置

```
lightsensor {
    compatible = "allwinner,stk3x1x";
    reg = <0x48>;
    device_type = "lightsensor";
    status = "okay";
    ls_twi_id = <0x1>;
    ls_twi_addr = <0x48>;
    ls_int = <&pio PH 4 GPIO_ACTIVE_LOW>;
    lightsensor-supply = <&reg_cldo3>;
};
```

说明

lightsensor 通常与 proximitysensor(距离感应) 一起，一般只需要配置 lightsensor 即可。

3.3.3 hallsensor 配置

```
hall_para {
    hall_name = "MH248";                    // 霍尔传感器名字
    status = "okay";                        // 标示使用霍尔传感器
    near_keycode = <142>;                   // 标示靠近时，上报的key值，不配置默认为SLEEP
    far_keycode = <143>;                    // 标示远离时，上报的key值，不配置默认为WAKEUP
    hall_int_port = <&r_pio PL 9 6 1 0xffffffff 0xffffffff>;// 霍尔传感器中断引脚
};
```

3.4 vibrator 通用配置

震动马达的通用配置如下。

```
motor_para {
    compatible = "allwinner,sunxi-vibrator";        // 对应驱动中的 .compatible
    motor_used = <1>;                       // 标示使用motor
    motor_shake = <0>;
    status = "okay";                        // 标示使能motor
    motor-supply = <&reg_ldoio1>;           // 标示motor使用哪路电源
    motor_ldo_voltage = <3300>;             // 标示motor的供电电压，单位mV
};
```

4 驱动适配

本章节主要讲述客户拿到 SDK 后，如何将自己的输入外设适配到 SDK 里，主要适用于挂载在 I2C 总线的设备。

4.1 TP 模组适配

拿到一个新的模组，通常有 2 种情况。

1. 模组适配性好，有完整 dts，及说明文档，这种模组直接参考原厂说明移植即可。
2. 较老的模组：支持性差，没有 dts 解析，电源域管理的模组，需要进行适配操作的。

这里的步骤主要基于上面的第 2 种情况进行适配，参考步骤如下。

4.1.1 dts 适配

根据硬件原理图，参考 CTP 通用配置章节，适配 dts，修改对应的引脚信息等。

```
ctp {
    compatible = "allwinner,gslX680";       //对应驱动中的compatible
    reg = <0x40>;                           //驱动i2c地址
    device_type = "ctp";
    status = "okay";
    ctp_name = "gslX680_3676_1280x800";
    ctp_twi_id = <0x0>;
    ctp_twi_addr = <0x40>;
    ctp_screen_max_x = <0x320>;
    ctp_screen_max_y = <0x500>;
    ctp_revert_x_flag = <1>;
    ctp_revert_y_flag = <1>;
    ctp_exchange_x_y_flag = <0x1>;
    ctp_int_port = <&pio PH 9 GPIO_ACTIVE_LOW>;
    ctp_wakeup = <&pio PH 10 GPIO_ACTIVE_LOW>;
    ctp-supply = <&reg_ldoio0>;
    ctp_power_ldo_vol = <3300>;
};
```

4.1.2 驱动适配

4.1.2.1 驱动编译

拿到一个驱动后，通常有如下文件。

```
├──── Kconfig
├──── Makefile
└──── tp.c     //驱动主程序
```

在 5.10 及以后的驱动版本中，TP 驱动文件通常放在如下路径。

```
bsp/drivers/input/ctp/
```

在以前的版本，则通常放在内核当中。

```
/drivers/input/touchscreen/
```

以 5.15 内核为例，驱动通常以文件夹形式独立放置，参考如下。

```
├──── ctp
   ├──── gslx680new       //gsl tp
   ├──── gt9xx           //Goodix gt系列tp
   ├──── Kconfig          //编译Kconfig
   ├──── Makefile         //编译的Makefile
   └──── wacom_pen
```

因此，适配一款新驱动时，通常如下。

1. 在 ctp 下创建一个新的文件夹，放置新的驱动文件，包含主程序，Makefile，Kconfig，其中 Makefile 及 Kconfig 应能正确编译出驱动。
2. ctp 文件夹下面的 Kconfig 及 Makefile 引入新驱动。

Kconfig 参考。

```
source "$(BSP_TOP)drivers/input/ctp/gt9xx/Kconfig"
```

Makefile 参考。

```
obj-$(CONFIG_AW_INPUT_CTP)   += gt9xx/
```

3. 在./build.sh menuconfig 中打开新的驱动的编译，然后通过./build.sh saveconfig 保存相关编译配置。

说明

目前 TP 均是采用编译成模块的方式，在系统启动后进行加载的。

4.1.2.2 compatible 适配

```
static const struct of_device_id gsl_of_match[] = {
    {.compatible = "allwinner,gslX680"},
    {},
};
```

说明

compatible 信息应与 dts 中配置完全一致。

4.1.2.3 使用 aw-input 框架适配驱动

引入 aw 的 input 框架，可以初始化驱动配置信息，协助管理电源域等。

设备初始化主要分为如下几步。

1. 创建一个全局的 ctp_config_info 结构体，其中，input_type 是必要的，用于初始化时获取对应的驱动信息。

```
struct ctp_config_info config_info = {
    .input_type = CTP_TYPE,
    .name = NULL,
};
```

2. 实现一个 start 函数，用于调用 aw-input 框架初始化函数。

```
static int startup(void)
{
    int ret = 0;
    pr_debug("*******************************************\n");
    if (input_sensor_startup(&(config_info.input_type))) {
        pr_err("%s: ctp_startup err.\n", __func__);
        return 0;
    } else {
        ret = input_sensor_init(&(config_info.input_type));
        if (ret != 0)
            pr_debug("%s:ctp_ops.init err.\n", __func__);
    }

    input_set_power_enable(&(config_info.input_type), 1);
    return 1;
}
```

start 函数的主要通用流程为：

- input_sensor_startup：获取对应的 dts 信息，并填充到 ctp_config_info 结构体。
- input_sensor_init：初始化一些信息，如电源域初始化，使用的引脚初始化等。
- input_set_power_enable：给外设上电。

3. 使用中断。

aw-input 支持中断注册，使用如下。

```
ret = input_request_int(&(config_info.input_type), gsl_ts_irq,
        CTP_IRQ_MODE, ts_init);
if (ret)
   printk("glsX680_init_events: request irq failed\n");
```

中断注册后，使用如下接口进行中断的使能。

```
input_set_int_enable(&(config_info.input_type), 0);
```

4. 电源域管理。

在驱动开发中，对模块使用的电进行管理是非常重要的，能够有效降低不必要的功耗。

开电和关电的接口使用如下：

```
input_set_power_enable(&(config_info.input_type), 0);
```

说明

对于没有固化 flash 的外设，在重新上电时，应注意将其重新初始化，刷写对应的寄存器信息等。

4.1.3 驱动加载

驱动的加载，通常是在启动完成后，在 Android 方案目录下，init.input.rc 文件里加载的，参考如下。

```
on property:sys.boot_completed=1
   #inmod tp module
   insmod /vendor/lib/modules/init-input.ko
   insmod /vendor/lib/modules/gslX680new.ko
   chown system system /sys/class/ctp/tp_idle
   chmod 0660 /sys/class/ctp/tp_idle
```

说明

TP 的加载应晚于 init-input 模块，该模块为 aw-input 框架驱动。

4.1.4 触摸方向适配

在驱动能够正常加载，且能够正常获得数据后，通常还需要对触摸方向进行适配。该工作应在默认显示方向配置完成后进行。

目前，为了兼容 dragonboard（竖屏）及 Android 横屏系统，我们使用了 dtbo 机制来使用 2 套 dts 控制 tp 的方向。

4.1.4.1 dtbo 介绍

dtbo 是 Android 系统使用的一种 dts overlay 机制，在 dtbo 中配置的 dts 信息会覆盖原有的 dts 信息。

简单的理解就是：dtbo 配置的方向作用于 Android 系统，dts 配置的方向作用于 dragonboard 及 linux。

在做方向配置时，应删除 dtbo 里的配置信息，因为 dtbo 无法通过 uboot 修改的方式进行调试。

4.1.4.2 方向配置

适配方向前，使用如下命令或者开发者选项中打开显示坐标点。

```
settings put system show_touches 1
settings put system pointer_location 1
```

适配步骤如下。

1. 设置 ctp_screen_max_x，ctp_screen_max_y，其中 x 默认为短边，y 默认为长边。
2. ctp_revert_x_flag，ctp_revert_y_flag 及 ctp_exchange_x_y_flag 均设置 0。
3. 触摸 4 个角，得到 xy 的 4 个角落坐标点。
4. 如果上下滑动变成了左右滑动，说明需要交换 xy，设置 ctp_exchange_x_y_flag 为 1。
5. 如果上滑变成了下滑，说明需要翻转 x，设置 ctp_revert_x_flag 为 1。
6. 如果左滑变成了右滑，说明需要翻转 y，设置 ctp_revert_y_flag 为 1。
7. 方向适配完成后，进行 xy 大小的适配即可。

参考上述步骤，多次测试，即可完成适配。适配完成后，如果需要使用 dtbo，则将 android 的方向配置放到 dtbo，dragonboard 的方向配置放到原来的 dts。

4.1.4.3 GSI 方向配置

针对 GSI 和正常固件横屏方向不一致需求的机器，参考例子：固件正常是横屏的，机器 GSI 是竖屏的，且需要在 gsi 机器上能够正常使用触摸的机器。

该需求应是：针对 GSI 完成 Android 方向的适配，完成后，在正常固件中，则使用 *.idc 文件对触摸进行旋转配置，参考例子。

```
touch.orientation=ORIENTATION_270
```

该代码表示将 TP 旋转 270 度。

idc 使用步骤如下。

1. 根据 getevent 里面的 tp 设备名，如 gslX680，在对应的 Android 方案目录里，创建 gslX680.idc 文件。

```
./input/gslX680.idc
```

2. 在 idc 里，设置需要的旋转角度，例如旋转 270 度

```
touch.orientation=ORIENTATION_270
```

3. 在 mk 里将对应的 idc 复制system/usr/idc/目录，目的是为了烧写 GSI 时，该文件会被刷掉。

```
PRODUCT_COPY_FILES += \
  $(LOCAL_MODULE_PATH)/gslX680.idc:$(TARGET_COPY_OUT_SYSTEM)/usr/idc/gslX680.idc
```

说明

idc 配置仅适用于 Android 13 及以上版本，旧版本使用ro.input_flinger.primary_touch.rotation属性。
具体可参考《Android_Rotation_ 使用指南》

4.2 GSL 模组使用

gsl-tp 模组是目前使用较多的模组，因此对于该模组，目前的相关 firmware config 是非常多的，当客户选择使用该 TP 后，对于该模组的适配步骤如下。

4.2.1 修改 dts 信息。

这里列举需要重点关注的信息。

```
ctp {
    compatible = "allwinner,gslX680";        //对应驱动中的compatible，需保持一致
    ctp_name = "gslX680_3676_1280x800";      //选择使用gsl的fw，与gslx680_fw_data里的名字一致
};
```

4.2.2 适配 firmware

客户拿到 gsl 驱动后，通常还会有一个 firmware 文件，格式是.h 类型的，里面是对应的 firmware 信息，也是需要做一些修改的。

firmware 文件里通常有 2 个数组。

```
unsigned int gsl_config_versions[]
static const struct fw_data
```

其中，第一个对应为该款 TP firmware 的版本信息及相关的外围数据，第二个为 fw_data 格式的 firmware 信息。

4.2.2.1 fw_data 数据

驱动中，使用一个 fw 结构体来保存已有的相关 firmware。

```
struct gslX680_fw_array {
   char name[64];
   unsigned int size;
   const struct fw_data *fw;
} gslx680_fw_grp;
```

结构体对应数据如下：

- name：对应 dts 配置中的 ctp_name，用于表示使用哪个 firmware。
- 表示 fw_data 里的数据大小，其大小为 fw 里的 fw_data 数组大小。
- 对应的 fw_data。

参考的使用案例如下。

```
struct gslX680_fw_array gslx680_fw_data[] = {
   {"gslX680_3676_855280", ARRAY_SIZE(GSLX680_FW_3676_855280), GSLX680_FW_3676_855280},
};
```

- “gslX680_3676_855280” 表示 ctp_name。
- GSLX680_FW_3676_855280：表示 gslX680_3676_855280 里的 fw_data 数据。

4.2.2.2 firmware version 数据

驱动中还使用一个数组，用来保存对应的 version 信息。

```
unsigned int *gslX680_fw_config_data[] = {
   gsl_config_data_id_3676_855280,
};
```

该数组用于保存各个 firmware 里面的 version 信息。注意数组里的顺序要与前面 fw_data 数据里的保持一致。

4.2.2.3 参考使用案例

1. dts 配置。

```
ctp {
      ctp_name = "gslX680_3676_855280";      //选择使用gslX680_3676_855280
};
```

2. 拿到的 firmware 文件信息。

```
static unsigned int gsl_config_data_id_3676_855280[] = {
//xxxxxxxxxxxxxxxxxxxx
}
static const struct fw_data GSLX680_FW_3676_855280[] = {
//xxxxxxxxxxxxxxxxxxxx
}
```

3.gslX680.c 配置信息

```
#include "gslX680_3676_855280.h

struct gslX680_fw_array gslx680_fw_data[] = {
  {"gslX680_3676_855280", ARRAY_SIZE(GSLX680_FW_3676_855280), GSLX680_FW_3676_855280},
};

unsigned int *gslX680_fw_config_data[] = {
  gsl_config_data_id_3676_855280,
};
```

4.2.3 Android firmware 制作节省内存占用

当使用的 firmware 多了以后，就容易出现一个驱动浪费内存的情况，为了针对这种情况，对 gslX680 驱动做了优化，将 firmware 放到用户空间，由驱动去读取 firmware，节省内存，我们提供了以下的配置和工具。

1. 在 android 中提供了 gsltool 这个工具用于生成驱动所能识别的 firmware，路径在 android/vendor/aw/public/package/bin/gsltool。
2. 有新的配置时可将配置头文件放入到 include 目录下，类似于以前的方式，修改 gsltool.c 文件，并在该目录下执行 mm 命令生成新的 gsltool，并执行gsltool firmware可生成所有的 firmware 文件，或者执行gsltool ${firmware_name} firmware，可生成单独的 firmware 文件。firmware 目录下是已支持的 firmware 文件。
3. 在方案配置目录下添加以下的配置：

```
$(call inherit-product, vendor/aw/public/package/bin/gsltool/gsl_firmware.mk)
```

调用到 gsl_firmware.mk，将指定或全部的 firmware 拷到指定的目录。

4. 可通过GSLFIRMWARELIST配置指定的 firmwares。

```
GSLFIRMWARELIST := gslX680_3676_1280x800
```

5. 如有 dragonboard 测试需求，还需要将生成的 bin 文件，放到如下目录

```
longan/test/dragonboard/src/testcases/gsltouch/firmware
```

4.3 NVT36XXX 模组使用

nvt36xxx 是 novatek 的一款 incell 屏幕，显示使用 MIPI 接口，TP 使用 SPI 接口。incell 屏的特点就是将显示及 TP 的控制器集成在了一起，使用同一路电源及中断、复位脚。所以在适配时需要格外注意与显示模块的同步及时序上的要求。

4.3.1 menuconfig 配置修改

```
CONFIG_TOUCHSCREEN_NT36xxx_HOSTDL_SPI=y
```

4.3.2 修改 dts 信息

由于是 SPI 接口的，所以其设备树配置需要放在 SPI 节点之下，同时需要打开 SPI 相关配置才可以正常使用。

```
novatek@0 {
    compatible = "novatek,NVT-ts-spi";
    reg = <0>;
    spi-max-frequency = <4800000>; // SPI通信速率，可选4800000/9600000/15000000/19200000
    novatek,irq-gpio = <&pio PH 9 GPIO_ACTIVE_LOW>;    // TP中断脚
    novatek,reset-gpio = <&pio PH 10 GPIO_ACTIVE_LOW>; // TP复位脚
    novatek,max-size-x = <1200>;   // x轴最大像素
    novatek,max-size-y = <1920>;   // y轴最大像素
    novatek,revert_x = <1>;       // 是否反转x轴坐标
    novatek,revert_y = <0>;       // 是否反转y轴坐标
    novatek,swap-x2y = <1>;       // 是否交换xy轴坐标
    //novatek,pen-support;        // 支持触摸笔
    status = "disabled";
};
```

说明

SPI 不在本文档讨论范围，具体详情与配置方法可参考《Linux_SPI_NG_开发指南》

4.3.3 修改 TP 宏配置

除了设备树配置外，驱动还有以下配置需要在源码里修改宏实现，可以视情况修改。

```
#define NVT_TOUCH_SUPPORT_HW_RST 1  // 是否支持硬件复位脚
#define WAKEUP_GESTURE 0        // 是否支持手势唤醒（暂不支持，需要硬件设计支持独立供电已保证休眠时TP不掉电）
#define TOUCH_MAX_FINGER_NUM 10    // 多点触摸支持的最多个数
```

4.3.4 适配 firmware

该款 TP 需要加载对应固件才可正常运行，且该固件在 TP 掉电或复位的情况下，都需要重新加载。为了方便使用，将固件直接集成进文件系统，启动后便可以自动加载。

在 android 配置文件（a523-y83/input/config.mk）中增加：

```
PRODUCT_COPY_FILES += \
  $(LOCAL_MODULE_PATH)/novatek_ts_fw.bin:$(TARGET_COPY_OUT_VENDOR)/etc/firmware/novatek_ts_fw.bin
```

在 input 启动文件（a523-y83/input/init.input.rc）中增加：

```
on property:sys.boot_completed=1
  insmod /vendor/lib/modules/nt36.ko
```

说明

该 TP 功能高度依赖 firmware 文件，若屏幕分辨率不一致或其他异常现象，请联系屏厂根据需求更新 firmware。

4.3.5 适配休眠唤醒机制

由于 incell 屏特性，电源与复位跟显示共用，在休眠唤醒时会进行掉电处理已保证设备功耗。因此在唤醒时，需要重新走上电流程，并在屏幕初始化完成后通知 TP 驱动唤醒并重新加载固件。

驱动中使用了内核显示框架的通知回调接口与显示驱动形成联系，并根据对应状态进行休眠或唤醒操作，具体代码如下：

```
ts->fb_notif.notifier_call = nvt_fb_notifier_callback;
ret = fb_register_client(&ts->fb_notif);
```

```
static int nvt_fb_notifier_callback(struct notifier_block *self, unsigned long event, void *data)
{
  struct fb_event *evdata = data;
  int *blank;
  struct nvt_ts_data *ts = container_of(self, struct nvt_ts_data, fb_notif);

  if (evdata && evdata->data && event == FB_EVENT_BLANK) {
    blank = evdata->data;
    if (*blank == 9) {
      nvt_ts_suspend(&ts->client->dev);
    } else if (*blank == 10) {
      nvt_ts_resume(&ts->client->dev);
    }
  }

  return 0;
}
```

说明

上面代码中 9/10 对应的事件需要从显示驱动中发出，具体可参考《Linux_LCD_ 开发指南》

4.4 Sensor 模组适配

Sensor 模组的适配，与 TP 一致，参考 TP 模组适配过程即可。

4.4.1 设备节点创建

对于 sensor 而言，必须有 enable 和 delay 节点，用于控制使能 sensor 以及设置采样率等。

对于多合一 sensor，如光感距离感应合一，应创建 2 个 input 设置，同时有不同的节点对不同 sensor 进行使能及设置采样率。

4.4.2 sensorhal 适配

sensorhal 的适配，请参考《Android Sensor 开发指南》。

4.5 keyboard 使用

如果是使用 lradc 作为 keyboard，可根据采用硬件的参数直接参考进行配置即可。

说明

在 linux 5.10 以后的内核版本，keyboard 为 sunxi-lradc 驱动，源码路径在 bsp/lradc，在 5.4 及以下版本，模块为 sunxi-keyboard，路径在 input 目录下。

4.6 vibrator 使用步骤

当前震动马达仅有供电配置，通电则震动，无相应震动强烈等级区分，因此使用步骤如下。

4.6.1 dts 配置

参照 vibrator 通用配置配置相应信息。

4.6.2 驱动编译

在 menuconfig 里配置编译 vibrator 驱动为 m。

```
--- a/configs/default/linux-5.15/bsp-defconfig
+++ b/configs/default/linux-5.15/bsp-defconfig
@@ -34,6 +34,7 @@ CONFIG_AW_MFD_AXP2101_I2C=y
 CONFIG_AW_REGULATOR_AXP2101=y
 CONFIG_AW_INPUT_SENSORINIT=y
 CONFIG_AW_INPUT_MISC=y
+CONFIG_AW_VIBRATOR=m
 CONFIG_AW_MMC=y
 # CONFIG_AW_DISP2 is not set
 CONFIG_AW_CRASHDUMP=y
```

图 4-1: vibrator defconfig

说明

部分版本为 SUNXI_VIBRATOR，具体以版本为准。

4.6.3 加载驱动

在系统启动完成后，加载 vibrator 驱动。

```
on property:sys.boot_completed=1
    insmod /vendor/lib/modules/sunxi-vibrator.ko
    chown system system /sys/class/vibrator/on
    chmod 0660 /sys/class/vibrator/on
```

其中，/sys/class/vibrator/on 为 vibrator 驱动的使能节点，用法为写入毫秒值，则震动对应的时间，最大不超过 5000，这里需要将其配置相应的权限。

4.6.4 使能 vibrator-hal

在 Android 方案目录，将 vibrator-hal 编译进去，参考如下。

```
# vibrator hal
PRODUCT_PACKAGES += \
    android.aw.hardware.vibrator-service
```

其中 vibrator hal 的源码位于 hardware/aw/vibrator/aidl。

4.6.4.1 驱动节点变更

如果使用的 vibrator 驱动节点与上述不一致，则需要变更节点，主要步骤如下。

1. 在加载驱动位置，修改对应的驱动节点，并配置权限。

2. 在 vibrator-hal 里，修改配置的节点路径，配置信息位于：hardware/aw/vibrator/aidl/Vibrator.cpp。

```
static const char THE_DEVICE[] = "/sys/class/vibrator/on";
```

4.7 霍尔传感器使用

当前霍尔感应用于唤醒，使用步骤如下。

4.7.1 dts 配置

参照 hallsensor 通用配置配置相应信息。

4.7.2 驱动编译

在 menuconfig 里配置编译 CONFIG_AW_HALL_SENSOR 驱动为 m。

说明

部分版本为 HALL_SENSOR，具体以版本为准。

4.7.3 加载驱动

在系统启动完成后，加载 vibrator 驱动。

```
on property:sys.boot_completed=1
    insmod /vendor/lib/modules/hall_sensor.ko
```

加载完成后，可以使用磁铁在霍尔传感器附近晃动进行测试，看功能是否正常。

5 dragonboard-input 测试

参考《Dragonboard 开发指南》。

6 调试

6.1 常用命令

常用的命令有 getevent，dumpsys input，dumpsys sensorservice 等。

6.1.1 getevent

getevent 使用详细如下。

```
getevent: invalid option --
Usage: getevent [-t] [-n] [-s switchmask] [-S] [-v [mask]] [-d] [-p] [-i] [-l] [-q] [-c count] [-r] [device]
  -t: show time stamps
  -n: don't print newlines
  -s: print switch states for given bits
  -S: print all switch states
  -v: verbosity mask (errs=1, dev=2, name=4, info=8, vers=16, pos. events=32, props=64)
  -d: show HID descriptor, if available
  -p: show possible events (errs, dev, name, pos. events)
  -i: show all device info and possible events
  -l: label event types and names in plain text
  -q: quiet (clear verbosity mask)
  -c: print given number of events then exit
  -r: print rate events are received
```

getevent 通常用于调试驱动是否有上报数据，查看驱动通路是否正常。

6.1.2 dumpsys input

dumpsys input 主要用于调试 TP，keyboard 等，获取当前对应的外设信息，最近的上报事件等，参考。

```
  5: gslX680
    Classes: TOUCH | TOUCH_MT
    Path: /dev/input/event4
    Enabled: true
    Descriptor: 7d7eeb71ea7765ef9584a43b6c32b201b90835d3
    Location:
    ControllerNumber: 0
    UniqueId:
    Identifier: bus=0x0018, vendor=0x0000, product=0x0000, version=0x0000
    KeyLayoutFile:
    KeyCharacterMapFile:
    ConfigurationFile: /system/usr/idc/gslX680.idc
```

```
  VideoDevice: <none>
```

上面可以看到具体配置的 idc 信息，以及 input 位置，是否使能等。

查看最近的 10 个上报事件。

```
RecentQueue: length=10
  MotionEvent(deviceId=5, eventTime=9774305876000, source=TOUCHSCREEN, displayId=0, action=MOVE,
      actionButton=0x00000000, flags=0x00000000, metaState=0x00000000, buttonState=0x00000000, classification=
      NONE, edgeFlags=0x00000000, xPrecision=1.6, yPrecision=0.6, xCursorPosition=nan, yCursorPosition=nan,
      pointers=[0: (102.4, 1001.9), 1: (135.5, 1017.9)]), policyFlags=0x62000000, age=5685ms
  MotionEvent(deviceId=5, eventTime=9774323603000, source=TOUCHSCREEN, displayId=0, action=MOVE,
      actionButton=0x00000000, flags=0x00000000, metaState=0x00000000, buttonState=0x00000000, classification=
      NONE, edgeFlags=0x00000000, xPrecision=1.6, yPrecision=0.6, xCursorPosition=nan, yCursorPosition=nan,
      pointers=[0: (102.4, 1001.9), 1: (135.5, 1017.9)]), policyFlags=0x62000000, age=5667ms
```

可看到事件类型，设备 id，接收事件的 displayid，以及上报的时间，坐标等。

6.1.3 dumpsys sensorservice

该命令主要用于调试 sensor，具体可见《Android Sensor 开发指南》。

6.1.4 打开画线显示触摸点

参考命令，或在开发者选项中打开即可。

```
settings put system show_touches 1
settings put system pointer_location 1
```

著作权声明

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明





（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。