全 志 科 技
Allwinner Technology

A20 平台 G-sensor 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	Initial version
V2.0	2013-06-17	1. 修改了读取 **sysconfig.fex** 的接口。 2. 增加了方向的配置方法说明。 3. 增加了自动检测 **gsensor** 时的相关说明。

目录

1. 前言

1.1. 编写目的

本文首先介绍了 G-sensor 模块的相关知识。

然后对已支持 G-sensor 在 A20 上的移植，进行总结归纳，旨在为大家提供一份详细的移植参考，从而获得统一的移植风格，以便于同事及客户的调试使用。

（本文以 A20 的 linux-3.4 中 bma250 驱动为参考，并不断进行更新补充中，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 G-sensor 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

本文主要分为三部分，

第一部分，第 2、3、4 章，主要介绍 G-sensor 模块的相关知识.如模块体系结构、数据结构等；

第二部分，第 5、6 章，介绍 G-sensor 的内核部分的移植及 android 部分的配置；

第三部分，第 7 章介绍 G-sensor 的相关调试方法。

2. 模块介绍

2. 1. 模块功能介绍

在人机交互过程中，G-sensor 起着非常重要的作用，gsensor 作为输入设备，能感知当前 G-sensor 传感器所处的空间状态，附着在 pad 上配合使用，能测量出 pad 在空间上的坐标状态，从而获知 pad 用户的操作意图：横竖屏切换，转弯等。

2. 2. 硬件介绍

目前 G-sensor 与 HOST 的连接有 4 个 pin 脚，分别为 VCC，GND，SDA,SCL。引脚正常工作时候的高电平均为 3.3V。

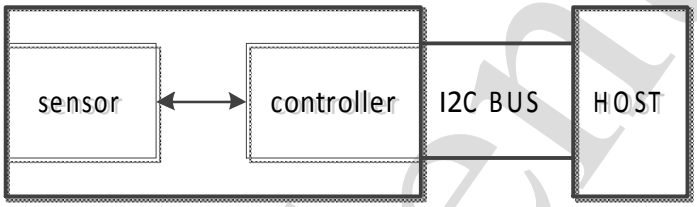


图 1 G-sensor 硬件连接图

G-sensor 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）、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. 3. 源码结构介绍

G-sensor 驱动源码位于如下两个目录中。

mma7660，mma8452，mma865x，afa750，kxtik 的源码位于 linux-3.4\drivers\hwmon 中。

bma250，lis3de，lis3dh，dmard10，mxc622x 的源码位于 linux-3.4\drivers\gsensor 中。

```
linux-3.4
drivers/hwmon/
├——— mma7660.c
├——— mma8452.c
├——— mma865x.c
├——— afa750.c
├——— kxtik.c
drivers/G-sensor/
├——— bma250.c
├——— lis3de_acc.c
```



```
├── lis3dh_acc.c
├── dmard10.c
├── mxc622x.c
```

2.4. 模块配置介绍

2.4.1. sys_config.fex 配置

（1）Gsensor 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 G-sensor 的配置项如下：

```
--------------------------------------------------------------------------------
 G sensor configuration
 gs_twi_id      ---    TWI ID for controlling G-sensor (0: TWI0, 1: TWI1, 2: TWI2)
--------------------------------------------------------------------------------
[G-sensor_para]
G-sensor_used            = 1
G-sensor_twi_id          = 2
G-sensor_int1            =
G-sensor_int2            =
```

文件配置说明如下：

配置项	配置项含义
G-sensor_used=xx	是否支持 G-sensor
G-sensor_twi_id =xx	I2C 的 BUS 控制选择，0：TWI0;1:TWI1;2:TWI2
G-sensor_int1＝xx	中断 1 的 GPIO 配置，目前暂不使用
G-sensor_int2=xx	中断 2 的 GPIO 配置，目前暂不使用

目前 G-sensor 采用轮询工作方式，在 sysconfig 中只需指定如下两个信息即可：

G-sensor_used 设为 1，代表启用 G-sensor；

G-sensor_twi_id 根据具体的电路，选择对应的 I2C；

（2）gsensor 自动检测配置

使用自动检测功能时，需要增加以下配置信息，增加设备的可选择性。

如果gsensor_used 设置为0 则将退出gsensor的自动检测。

```
;--------------------------------------------------------------------------------
; G sensor automatic detection configuration
```

```
;gsensor_detect_used   --- Whether startup automatic inspection function. 1:used,0:unused
;Module name postposition 1 said detection, 0 means no detection.
;----------------------------------------------------------------------------
[gsensor_list_para]
gsensor_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。
bma250                  = 1 //设置为 1，该模块支持的 I2C 地址添加到扫描列表
mma8452                 = 1
mma7660                 = 1
mma865x                 = 1
afa750                  = 1
lis3de_acc              = 1
lis3dh_acc              = 1
Kxtik                   = 1
dmard10                 = 0 //设置为 0，该模块支持的 I2C 地址从扫描列表中剔除
dmard06                 = 1
mxc622x                 = 1
fxos8700                = 0
lsm303d                 = 0
```

当gsensor_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。

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

2.4.2. menuconfig 的配置

对于 G-sensor 的的内核配置，可通过命令 make ARCH=arm menuconfig 进入配置主界面，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 2 所示：

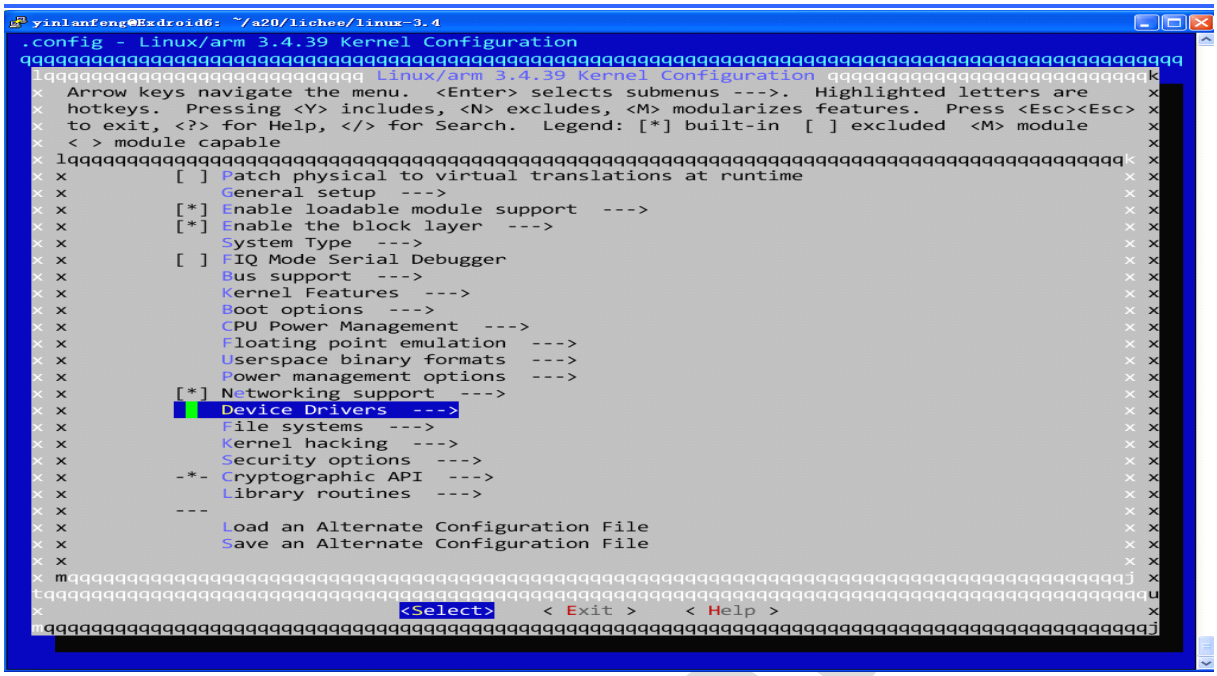


图 2 :Device Drivers 选项配置

进入 Device Drivers 配置后，如果需要配置 mma7660，mma8452，mma865x，afa750 等，选择 Hardware Monitoring support，如图 3 所示：

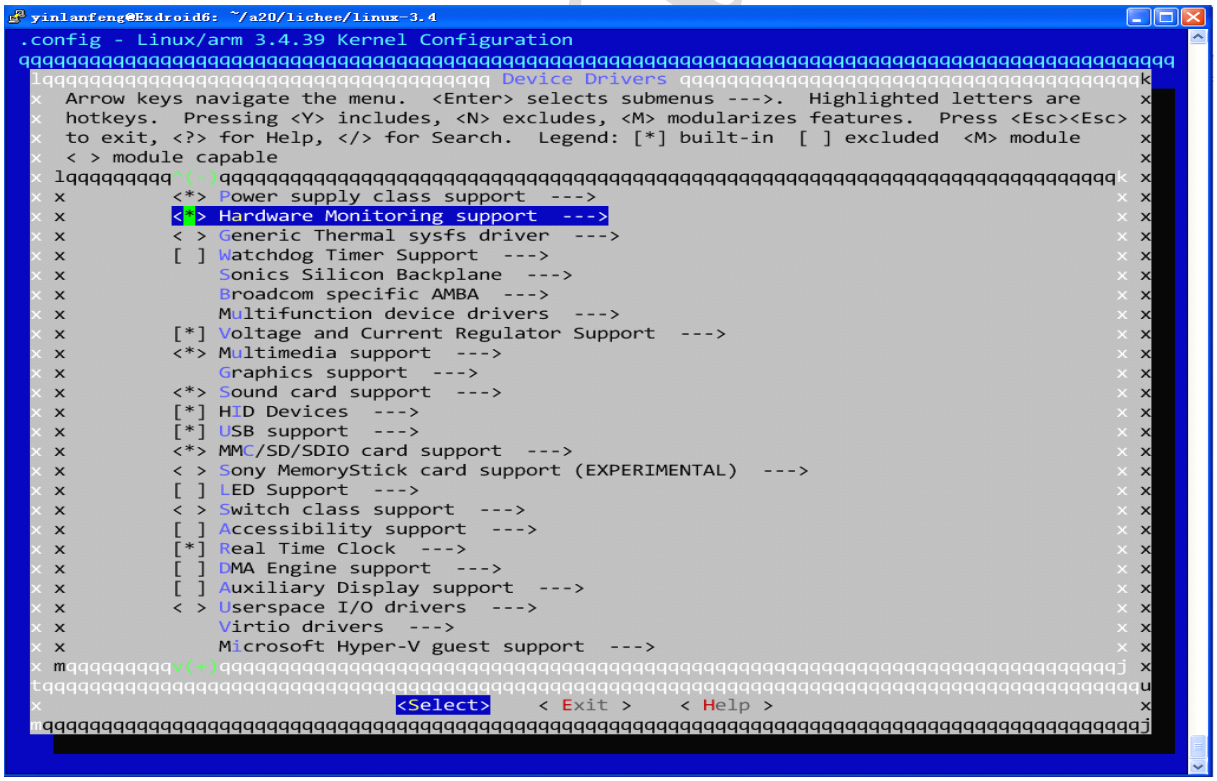


图 3 :Hardware Monitoring support 选项配置

Hardware Monitoring support 选项配置下的驱动如下图所示：

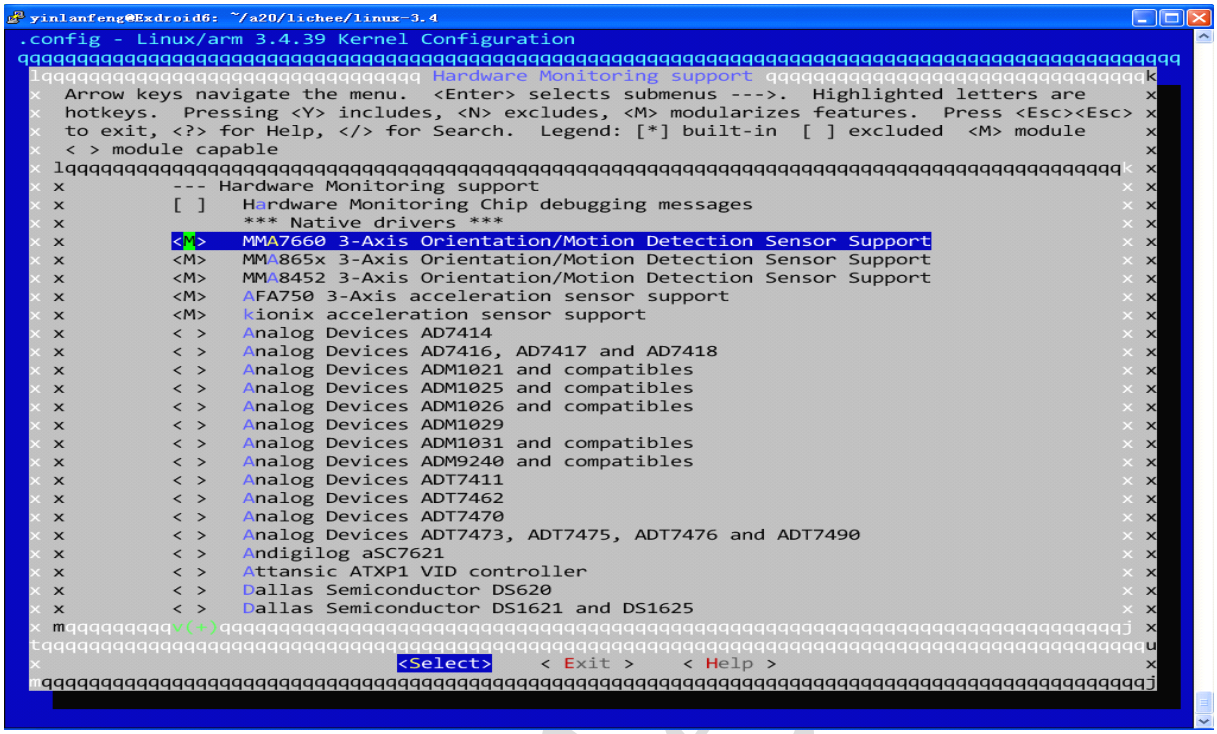


图 4 :Hardware Monitoring support 设备驱动

进入 Device Drivers 配置后，如果需要配置 bma250，lis3de，lis3dh 等，选择 G-sensor support，如图 5 所示：

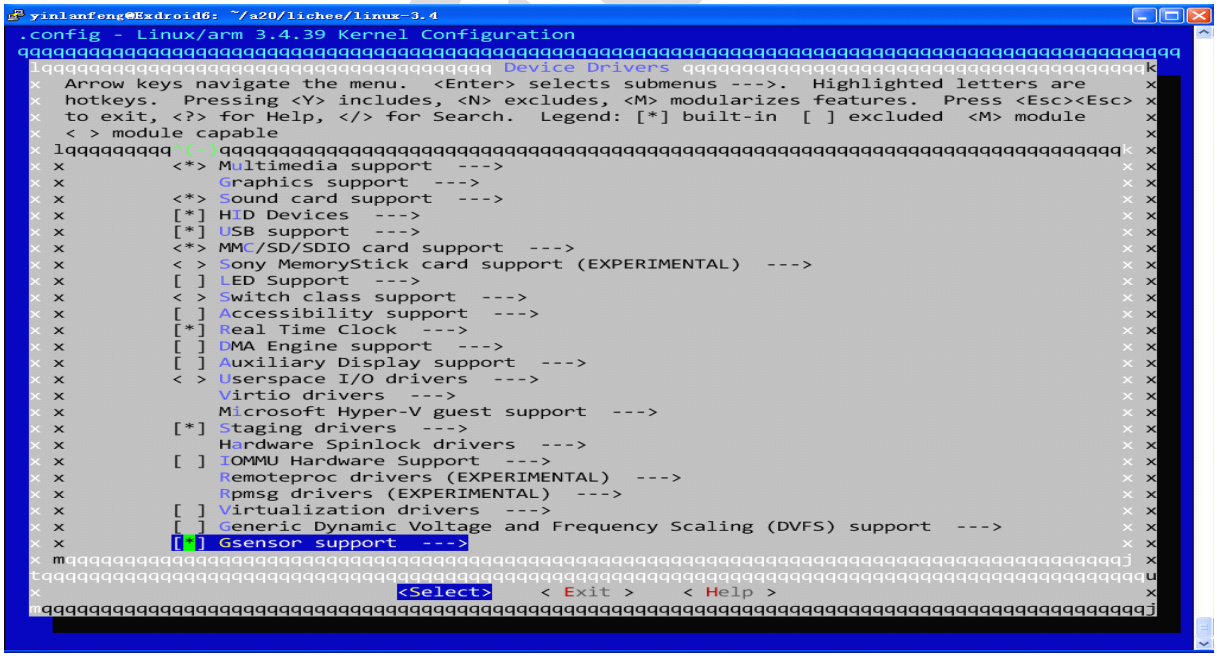


图 5 :I2C G-sensor support 选项配置

最后，选择相应的 G-sensor 模块驱动选项，可选择直接编译进内核，也可以选择编译成模块。如图 6 所示：

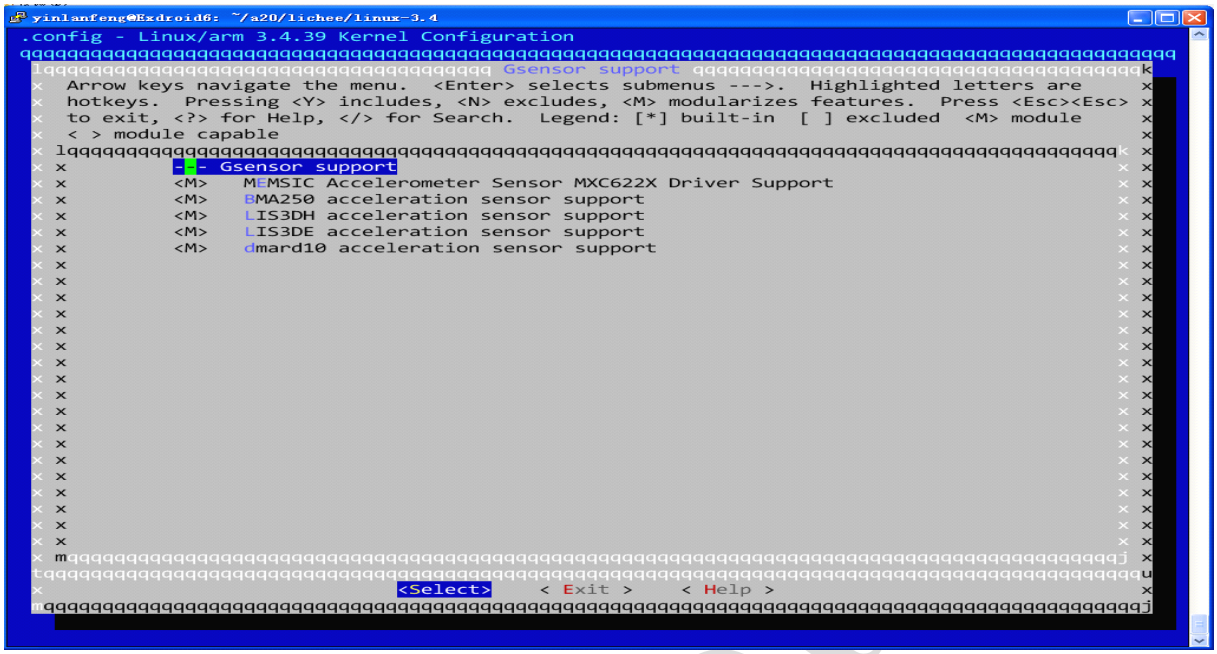


图 6 G-sensor 驱动模块选项配置

3. 模块体系结构描述

G-sensor 模块的体系结构图，如图 6 所示。

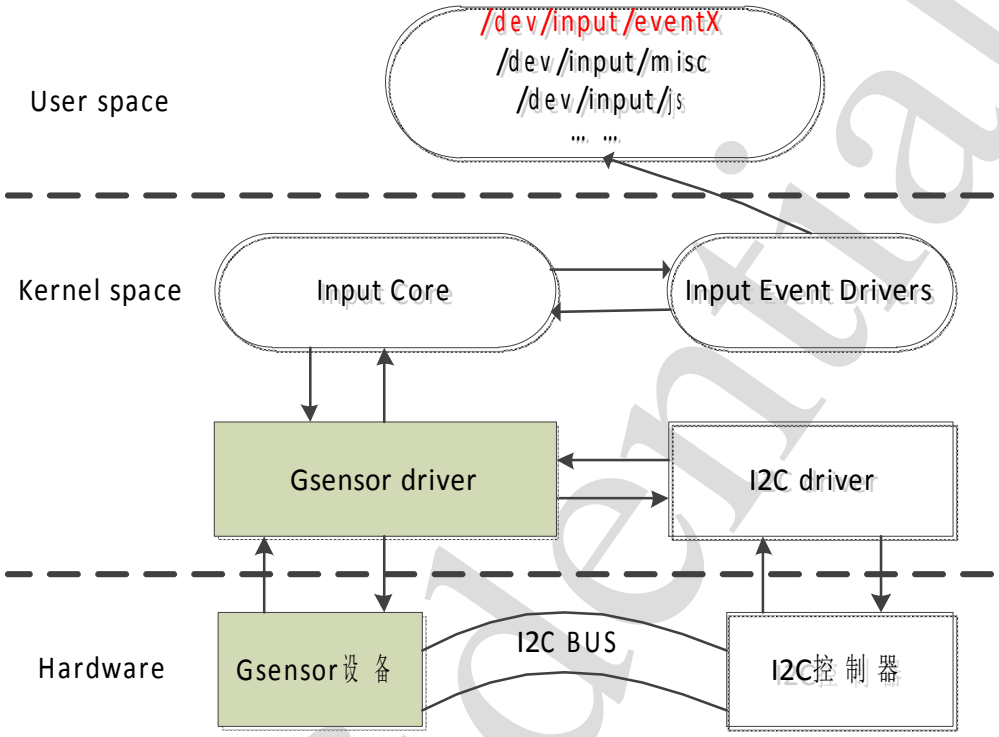


图 6:G-sensor 模块体系结构图

G-sensor 设备为使用 I2C 总线进行通信的输入设备，G-sensor driver 通过调用 I2C 驱动的相应接口来实现对 G-sensor 设备的控制、通信，如 G-sensor driver 对 G-sensor 设备硬件各寄存器的读写访问等。

G-sensor driver 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 G-sensor 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。

4. 模块数据结构描述

4.1. struct i2c_driver bma250_driver

bma250_driver：该变量会注册到 i2c_driver 中，driver.name 为匹配设备名，probe 为设备的侦测函数，address_list 为 I2C 的 scan 地址，suspend 与 resume 为注册进内核的休眠唤醒函数。

```
static struct i2c_driver bma250_driver = {
    .class = I2C_CLASS_HWMON,
    .driver = {
        .owner    = THIS_MODULE,
        .name     = SENSOR_NAME,
    },
    .id_table  = bma250_id,
    .probe          = bma250_probe,
    .remove         = bma250_remove,
#ifdef CONFIG_HAS_EARLYSUSPEND
#else
#ifdef CONFIG_PM
    .suspend = bma250_suspend,
    .resume = bma250_resume,
#endif
#endif
    .address_list    = normal_i2c,
};
```

4.2. struct bma250_data

struct bma250_data :代表了 G-sensor 驱动所需要的信息的集合，用于帮助实现对采样信息的处理。

```
struct bma250_data {
    struct i2c_client *bma250_client;
    atomic_t delay;
    atomic_t enable;
    unsigned char mode;
    struct input_dev *input;
```

```
    struct bma250acc value;
    struct mutex value_mutex;
    struct mutex enable_mutex;
    struct mutex mode_mutex;
    struct delayed_work work;
    struct work_struct irq_work;
#ifdef CONFIG_HAS_EARLYSUSPEND
    struct early_suspend early_suspend;
    unsigned char range_state;
    unsigned char bandwidth_state;
#endif
};
```

4.3. struct bma250acc

struct bma250acc 用于记录采样时获得的 x 轴，y 轴，z 轴的坐标信息。

```
struct bma250acc{
    s16  x,
    y,
    z;
} ;
```

4.4. struct sensor_config_info

struct sensor_config_info 用于记录读取到的 sysconfig.fex 中 gsensor 的相关信息。
input_type 为 gsensor 设备的类型，gsensor 使用 GSENSOR_TYPE。
sensor_used 为 sysconfig.fex 中的 gsensor_used 值。
twi_id 为 sysconfig.fex 中的 gsensor_twi_id 值。

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

5. 模块移植 demo

5.1. G-sensor 驱动概述

G-sensor 驱动作为硬件与软件的一个桥梁，实现对 G-sensor 控制器硬件初始化，获取 G-sensor 控制器采集到的位置坐标信息（必要时，对数据进行滤波和用户操作意图识别），上报用户操作相关信息于操作系统，经上层系统处理后，正确响应用户的意图。

G-sensor 驱动在系统中必须满足如下要求：

- 接口：G-sensor 驱动，不应自行决定是否上报，上报频率等，应提供接口，供上层应用控制驱动的运行和数据上报：包括使能控制 Enable， 上报时延 delay 等；通常通过 sys 文件系统提供，这部分实现，遵循标准的 linux 规范；
- 上报数据的方式：或者提供接口供上层访问（eg: ioctl），或者挂接在 input 系统子系统上，使用 input 系统子系统的接口，供上层使用(eg: input core)；

5.2. G-sensor 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，detect 函数，suspend 以及 resume 函数等。以下将以 bma250 系列驱动移植过程为例进行说明。源码路径为：\lichee\linux-3.4\drivers\G-sensor\bma250.c

5.2.1. 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，G-sensor 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：

```
struct input_dev *input_allocate_device(void)
```

（2）注册输入设备，并和对应的 handler 处理函数挂钩：

```
int input_register_device(struct input_dev *dev)
```

（3）注销 Input 设备：

```
int input_unregister_device(struct input_dev *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）上报坐标结束时同步信号：

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

5.2.2. I2C 设备关键接口

目前 G-sensor 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。在 bma250.c 驱动中的 i2c_driver 变量为 bma250_driver，其具体定义在 4.1 节中已经给出。

（4）i2c_del_driver

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

（5）sysfs_create_group

sysfs 接口，使用如下格式进行创建：

```
sysfs_create_group(&pdev->dev.kobj, &dev_attr_grp);
```

5.2.3. 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
debug_mask	设置打印等级变量	在 module_para_name 中使用
normal_i2c	I2C 总线扫描地址数组	i2c_driver 结构体中，注册设备地址
i2c_address	I2C 总线 detect 函数扫描地址数组	gsensor_detect 函数扫描地址

5.2.4. 需要包含的头文件

Gsensor 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 sensor_config_info 结构体以及读取 sysconfig.fex 的相关调用接口与打印等级的定义。

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”

5.2.5. I2C 地址

在 G-sensor 驱动中 I2C 地址有四个，均放置在扫描地址数组 normal_i2c。

normal_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[] = {0x18,0x19,0x38,0x08,I2C_CLIENT_END};
```

5.2.6. detect 函数

G-sensor 驱动中的 gsensor_detect 函数实现硬件检测功能，它会遍历此驱动支持所有芯片的 I2C 地址，如果某个 I2C 地址能够正常通信并读到了正确的 chip_id，则 dectect 成功，继续加载驱动。否则此驱动支持的芯片没有链接在电路中，驱动模块加载失败。bma_250 的 detect 函数如下。

```
static int gsensor_detect(struct i2c_client *client, struct i2c_board_info *info)
{
    struct i2c_adapter *adapter = client->adapter;
    int ret , i = 0 ,retry = 2;
    if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_BYTE_DATA))
        return -ENODEV;
    if (twi_id == adapter->nr) {
     while(retry--) {
        for (i2c_num = 0; i2c_num < (sizeof(i2c_address)/sizeof(i2c_address[0]));i2c_num++) {
            i = 0;
            client->addr = i2c_address[i2c_num];
            ret = i2c_smbus_read_byte_data(client,BMA250_CHIP_ID_REG);
            dprintk(DEBUG_INIT, "%s:addr = 0x%x, i2c_num:%d, Read ID value is :%d\n",
                                __func__, client->addr, i2c_num, ret);
            while((chip_id_value[i++]) && (i < 5)){
                    dprintk(DEBUG_INIT, "chip:%d\n", chip_id_value[i - 1]);
                     if((ret & 0x00FF) == chip_id_value[i - 1]){
                            strlcpy(info->type, SENSOR_NAME, I2C_NAME_SIZE);
                                    return 0;
                     }
             }
        }
    }
        dprintk(DEBUG_INIT, "%s:Bosch Sensortec Device not found,\
                maybe the other gsensor equipment! \n",__func__);
        return -ENODEV;
```

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

i2c_address 数组存储了此驱动支持的所有的 I2C 地址，对每个 I2C 地址读取 chip_id，如果读取成功，且为驱动支持的 chip_id，则 detect 成功。

如果设备没有 chip id 的可以进行 i2c 检测设备是否存在。如下所示：以 mma7660.c de detect 为例子如下所示：

```
static int gsensor_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(config_info.twi_id == adapter->nr){
            dprintk(DEBUG_INIT, "%s: addr= %x\n",__func__,client->addr);
            ret = gsensor_i2c_test(client);
            if(!ret){
                dprintk(DEBUG_INIT, "%s:I2C connection might be something wrong or\
                            maybe the other gsensor equipment! \n",__func__);
                return -ENODEV;
            }else{
                        strlcpy(info->type, SENSOR_NAME, I2C_NAME_SIZE);
                    return 0;
            }
    }else{
        return -ENODEV;
    }
}
```

5.2.7. G-sensor 驱动 init 函数

该函数中主要任务为调用 input_fetch_sysconfig_para 函数获取 sys_config.fex 中的配置信息；接着将 gsensor_detect 函数赋值给 i2c_driver，调用接口注册 i2c 设备驱动。如下所示：

```
static int __init BMA250_init(void)
{
    int ret = -1;
    dprintk(DEBUG_INIT, "bma250: init\n");
    if(input_fetch_sysconfig_para(&(config_info.input_type))){
        printk("%s: err.\n", __func__);
        return -1;
    }
    if(config_info.sensor_used == 0){
        printk("*** used set to 0 !\n");
        printk("*** if use sensor,please put the sys_config.fex gsensor_used set to 1. \n");
        return 0;
    }
    bma250_driver.detect = gsensor_detect;
    ret = i2c_add_driver(&bma250_driver);
    return ret;
}
```

5.2.8. super standby 支持

（1）super standby 中 suspend 以及 resume 的处理

super standby 就是关掉除 AVCC 和 DRAM_VCC 电源以外的所有电源，因此在休眠时 gsensor 会被断电，在 resume 时，需要重新初始化 gsensor 的一些寄存器。

以 bma250 为例，其 suspend 函数及 resume 函数如下。在 suspend 时判断休眠类型是否为 super standby，如果是，则保存当前 gsensor 的一些寄存器信息。在 resume 时，判断是否为 super standby，如果是，则将 suspend 时保存的寄存器信息重新写入 gsensor。

```
static void bma250_early_suspend(struct early_suspend *h)
{
    struct bma250_data *data = container_of(h, struct bma250_data, early_suspend);
    dprintk(DEBUG_SUSPEND, "bma250: early suspend\n");
    if (NORMAL_STANDBY == standby_type) {
        ……
    } else if (SUPER_STANDBY == standby_type) {
        if (bma250_get_bandwidth(data->bma250_client, &data->bandwidth_state) < 0)
```

```
            printk("suspend: read bandwidth err\n");
        if (bma250_get_range(data->bma250_client, &data->range_state) < 0)
            printk("suspend: read range err\n");
        ……
    }
}

static void bma250_late_resume(struct early_suspend *h)
{
    struct bma250_data *data = container_of(h, struct bma250_data, early_suspend);
    dprintk(DEBUG_SUSPEND, "bma250: late resume\n");
    if (NORMAL_STANDBY == standby_type) {
        ……
    } else if (SUPER_STANDBY == standby_type) {
        if (bma250_set_bandwidth(data->bma250_client,
                    data->bandwidth_state) < 0)
            printk("suspend: write bandwidth err\n");
        if (bma250_set_range(data->bma250_client, data->range_state) < 0)
            printk("suspend: write range err\n");
        ……
    }
}
```

（2）使用 input_polled_dev 的设备 suspend 以及 resume 需要关注点

gsensor 驱动注册为 input_polled_dev，此类驱动的轮询延时工作队列为 input-polldev.c 自动管理的。在休眠时，此轮询延时工作队列并未关闭，会导致系统无法进入休眠。如 mma7660, mma8452, mma865x 等。这些驱动需要在 suspend 函数自己去关需要调用函数：

```
input_polled_dev->input->close(struct input_dev *dev)
```

将设备关闭，否则将可以导致设备无法进入休眠。以 mma8452 为例子进行说明。字体加粗部分即为打开与关闭轮询延时工作队列的函数调用。

```
static void mma8452_early_suspend(struct early_suspend *h)
{
    ……
    mutex_lock(&enable_mutex);
```

```
    atomic_set(&mma8452_suspend_id, 1);
    mma8452_idev->input->close(mma8452_idev->input);
    mutex_unlock(&enable_mutex);
    ……
}

static void mma8452_late_resume(struct early_suspend *h) //(struct i2c_client *client)
{
        ……
        mutex_lock(&enable_mutex);
        atomic_set(&mma8452_suspend_id, 0);
        mma8452_idev->input->open(mma8452_idev->input);
        mutex_unlock(&enable_mutex);
    ……
}
```

5.2.9. 模块加载及 resume 延时优化

有些 G-sensor 在初始化时，初始化寄存器后，需要一定的延时才能正常工作。一般这个延时都是忙等待，这样在加载驱动及 super standby 唤醒时，相当耗时，影响用户体验。现在进行了优化。思路是采用工作队列线程将寄存器初始化部分单独延后执行，不影响主进程的运行。以 mma8452 为例，它在初始化寄存器时有一个 100ms 的延时。因此，在其休眠唤醒时的 resume 函数中进行了相应的优化，其源码如下，在休眠唤醒时 resume 函数如下。

```
static void mma8452_late_resume(struct early_suspend *h) //(struct i2c_client *client)
{
    int result;

    dprintk(DEBUG_SUSPEND, "mma8452 late resume");

    if (NORMAL_STANDBY == standby_type) {
        ……
    } else if (SUPER_STANDBY == standby_type) {
        queue_work(mma8452_resume_wq, &mma8452_resume_work);
    }
```

```
    dprintk(DEBUG_SUSPEND, "mma8452 late resume end");
    return ;
}
```

字体加黑的部分为启动工作队列线程来将 mma8452 的寄存器初始化延后执行。mma8452_resume_work 对应的执行函数为 mma8452_resume_events，其源码如下。

```
static void mma8452_resume_events (struct work_struct *work)
{
    mma8452_init_client(mma8452_i2c_client);
    ……
}
```

mma8452_init_client 为初始化 mma8452 寄存器函数，其中有一个 100ms 的忙等待延时。这部分的总体实现请参考 mma8452.c 源码。

5.2.10. 模块 remove 函数 check

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。

5.2.11. Sysfs 接口

Gsensor 驱动中，需要提供使能控制以及上报时延等基本接口，通常通过 sysfs 文件系统提供，sensors 的驱动中，使能控制统一命名为 enable，上报时延为 delay，如果名字变换，将可能造成 hal 层将无法向驱动层中写入使能控制以及时延，造成设备不工作或者是数据上报慢，影响体验效果。

（1）函数宏 DEVICE_ATTR

DEVICE_ATTR 宏声明有四个参数，分别是名称、权限位、读函数、写函数。其中读函数和写函数是读写功能函数的函数名。需要注册的使能控制的名称应该设置为 **enable**；上报时延的名称应该设置为 **delay**。基本函数为 enable 以及 delay，如需要进行相关的调试，可以增加其他的接口。如下所示：

```
……
static DEVICE_ATTR(delay, S_IRUGO|S_IWUSR|S_IWGRP,
        bma250_delay_show, bma250_delay_store);
static DEVICE_ATTR(enable, S_IRUGO|S_IWUSR|S_IWGRP,
        bma250_enable_show, bma250_enable_store);
```

```
.....
```

（2）attribute 结构体的填充

将 DEVICE_ATTR 声明的接口填充到 attribute 结构体中，如下所示：

```
static struct attribute *bma250_attributes[] = {
    .....
    &dev_attr_delay.attr,
    &dev_attr_enable.attr,
    NULL
};
```

该结构体必须以 NULL 结束。

（3）attribute_group 结构体的填充

完成了 attribute 结构体的填充后，将实现 attribute_group 结构体的填充，如下所示：

```
static struct attribute_group bma250_attribute_group = {
    .attrs = bma250_attributes
};
```

（4）sysfs 接口的注册

调用系统提供函数，进行注册，如下：

```
err = sysfs_create_group(&data->input->dev.kobj,
                         &bma250_attribute_group);
```

5.2.12. Kconfig 以 Makefile 文件

添加一个新的 G-sensor 驱动需要修改 Kconfig 文件和 Makefile 文件以使得能够在 menuconfig 中选中 G-sensor 驱动并编译生成模块或者是直接编译进内核。现 bma250.c 为例，将源码拷贝到目录: ..\exdroid\lichee\linux-3.4\drivers\gsensor 下，按照文件 Kconfig 与 Makefile 的形式，将 G-sensor 的驱动注册进去，如下：

Kconfig 文件：

```
config SENSORS_BMA250
    tristate "BMA250 acceleration sensor support"
    depends on I2C
    help
      If you say yes here you get support for Bosch Sensortec's
      acceleration sensors BMA250
```

Makefile 文件：

```
obj-$(CONFIG_SENSORS_BMA250)        += bma250.o
```

两个文件中设置的名字必须一致，即“CONFIG_SENSORS_BMA250”需一致，编译之后会自动的生成.Ko 文件。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 workspace\exdroid\lichee\linux-3.4 上，输入命令：

```
make ARCH=arm menuconfig
```

进入目录 Device Drivers\gsensor support 即可看到添加的驱动。

5.2.12. 驱动调试信息

为了实现动态开关打印信息，在 input 子系统的驱动模块中加入一个模块参数 debug_mask。驱动中所有的打印 printk 前加一个 if 条件判断语句，进行打印等级与 debug_mask 的按位与，debug_mask 相应等级位为真（1），则打印此 printk，为假（0）则关闭此打印。

在 G-sensor 代码 bma250.c 中，打印消息设置如下。

在文件开头，首先定义打印等级，模块参数，及打印函数，示例代码如下：

```
static u32 debug_mask = 0;
#define dprintk(level_mask, fmt, arg...) if (unlikely(debug_mask & level_mask)) \
    printk(KERN_DEBUG fmt , ## arg)

module_param_named(debug_mask, debug_mask, int, S_IRUGO | S_IWUSR | S_IWGRP);
```

如代码中所示，枚举定义了打印等级，每一个等级占用一位。

debug_mask 为控制打印的模块参数，默认为 0，关闭所有的打印。

dprintk 函数为封装好的打印函数，在驱动中调用它来打印相关的调试信息，如下面代码中所示。

```
static void bma250_early_suspend(struct early_suspend *h)
{
    struct bma250_data *data =
        container_of(h, struct bma250_data, early_suspend);

    dprintk(DEBUG_SUSPEND, "bma250: early suspend\n");
    ……
}
```

在 bma250_early_suspend 函数中调用 dprintk 函数打印调试信息，打印等级设为 DEBUG_SUSPEND。当 debug_mask 的第 3 位为 1，则此条打印消息会成功打印；当 debug_mask 的第 3 位为 0，则屏蔽此打印消息。

在具体的驱动调试中，如果驱动编译成模块，则在驱动加载时，设置 debug_mask 的值，从而实

现对各级打印消息的开关控制。

```
insmod /system/vendor/modules/bma250.ko debug_mask=0x0b
```

此方法存在的问题：当驱动加载不成功时，没有模块参数文件节点来让我们修改 debug_mask 的值，为了避免这个不足，在初始化函数 init 及 probe 函数中的初始化 fail 的相关打印最好写成 printk 直接打印，这些 printk 在模块正常工作时是不会被打印的，只有模块加载失败时才会被打印出来。

6. G-sensor Android 层配置

添加 G-sensor 驱动模块对应的 hal 层。G-sensor 的 hal 层源码为 android\device\softwinner\fiber-common\hardware\libhardware\libsensors\目录下的。

6.1 方向的配置

同时，在 android\device\softwinner\wingr-xxx 目录下有个 gsensor.cfg 文件，记录了如下几个变量。

gsensor_name	G-sensor 名称，必须与驱动中设备名相同
gsensor_direct_x	G-sensor x 轴的方向，当定义成 true 时，x 轴取正值，当定义为 false 时，x 轴取负值
gsensor_direct_y	G-sensor y 轴的方向，当定义成 true 时，y 轴取正值，当定义为 false 时，y 轴取负值
gsensor_direct_z	G-sensor z 轴的方向，当定义成 true 时，z 轴取正值，当定义为 false 时，z 轴取负值
gsensor_xy_revert	XY 轴对调，当设为 TRUE 时，x 轴变为原来 y 轴

因为最终产品电路设计可能不一样，导致 G-sensor 可能贴的方向不一样，所以会出现添加 G-sensor 后，用户实际使用中会出现 G-sensor 方向不对的现象。需要根据以下步骤进行方向的调试。

Gsensor 方向调试说明：

假定机器的长轴为 X 轴，短轴为 Y 轴，垂直方向为 Z 轴。

首先调试 Z 轴：

第一步观察现象：

旋转机器，发现当只有垂直 90°时或者是在旋转后需要抖动一下，方向才会发生变化，则说明 Z 轴反了。若当机器大概 45°拿着的时候也可以旋转，说明 Z 轴方向正确。无需修改 Z 轴方向。

第二步修改 Z 轴为正确方向。

此时需要找到当前使用模组的方向向量（根据模组的名称）。如果此时该方向 Z 轴向量（gsnesor_direct_z）的值为 false，则需要修改为 true；当为 true，则需要修改为 false。通过 adb shell 将修改后的 gsnesor.cfg 文件 push 到 system/usr 下，重启机器，按第一步观察现象。

其次查看 X，Y 轴是否互换：

第一步观察现象：

首先假定长轴为 X 轴，短轴为 Y 轴，以 X 轴为底边将机器立起来。查看机器的 X，Y 方向是否正好互换，若此时机器的 X，Y 方向正好互换，在说明需要将 X，Y 方向交换。若此时 X，Y 方向没有反置，则进入 X，Y 方向的调试。

第二步 交换 X，Y 方向

当需要 X，Y 方向交换时，此时需要找到当前使用模组的方向向量（根据模组的名称）。如果此

时该 X，Y 轴互换向量（gsensor_xy_revert）的值为 false，则需要修改为 true，当为 true，则需要修改为 false。通过 adb shell 将修改后的 gsnesor.cfg 文件 push 到 system/usr 下，重启机器，按第一步观察现象。

再次调试 X，Y 轴方向：

第一步观察现象：

首先假定长轴为 X 轴，短轴为 Y 轴，以 X 轴为底边将机器立起来，查看机器的方向是否正确，如果正确，说明长轴配置正确，如果方向正好相反，说明长轴配置错误。将机器旋转到短轴，查看机器方向是否正确，如果正确，说明短轴配置正确，如果方向正好相反，说明短轴配置错误。

第二步修改 X，Y 轴方向：

当需要修改 X，Y 轴方向时，当只有长轴方向相反或者是只有短轴方向相反时，则只修改方向不正确的一个轴，当两个方向都相反时，则同时修改 X 与 Y 轴方向向量。找到当前使用模组的方向向量（根据模组的名称）。

若长轴方向相反，如果此时该方向 X 轴向量（gsnesor_direct_x）的值为 false，则需要修改为 true，当为 true，则需要修改为 false。

若短轴方向相反，如果此时该方向 Y 轴向量（gsnesor_direct_y）的值为 false，则需要修改为 true，当为 true，则需要修改为 false。

通过 adb shell 将修改后的 gsnesor.cfg 文件 push 到 system/usr 下，重启机器，按第一步观察现象。若发现还是反向 X 轴或者 Y 轴的方向仍然相反，则说明 X 轴为短轴，Y 轴为长轴。此时：

若长轴方向相反，如果此时该方向 Y 轴向量（gsnesor_direct_y）的值为 false，则需要修改为 true，当为 true，则需要修改为 false。

若短轴方向相反，如果此时该方向 X 轴向量（gsnesor_direct_x）的值为 false，则需要修改为 true，当为 true，则需要修改为 false。

6.2 驱动的加载

在 android 上移植 G-sensor 还必须将驱动拷贝到 android 打包目录的对应文件夹中，并在 init.sunxi 中将其加载。具体修改方法如下：

（1）、android4.x.x

android4.x.x 会在 extract-bsp 的时候将其拷贝到对应目录。

android4.x.x 对应的目录已经更改为：$PRODUCT_ROOT/system/vendor/modules/

（2）、修改 init.sunxi.rc 文件，android4.x.x 修改 init.sun7i.rc,在对应的文件中添加：insmod /vendor/modules/bma250.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。

7. 模块调试

7.1 调试信息的使用方法

(1)驱动加载时，设置打印等级

在驱动编译为模块的情况下，可以在驱动加载时，设置 debug_mask 的值，从而实现对各级打印消息的开关控制。如下所示

```
insmod /system/vendor/modules/bma250.ko debug_mask=0x0b
```

(2)通过 shell 将打印打开

通过 shell 将打印打开步骤如下：

一、登录到 shell 界面（adb shell 或者是串口调试等）

二、进入模块节点，使用命令：

```
root@android:/ # cd sys/module/bma250/parameters
```

三、查看模块参数名称

```
root@android:/sys/module/bma250/parameters # ls
ls
debug_mask
```

四、查看模块当前参数值

```
root@android:/sys/module/bma250/parameters # cat debug_mask
cat debug_mask
0
```

五、设置模块参数值

```
root@android:/sys/module/bma250/parameters # echo 0x03 > debug_mask
echo 0x03 > debug_mask
```

六、查看设置是否成功

```
root@android:/sys/module/bma250/parameters # cat debug_mask
cat debug_mask
3
```

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。

7.2 gsensor 驱动调试步骤

（1）确保硬件各个管脚的连接顺序正确；

（2）上电，测试各个管脚信号的电压正常，只有在保证硬件正常的情况下，进行软件驱动调试，方可保证驱动能够正常工作（该处最容易被很多软件开发人员忽视，务必注意，方可节省大部分时间）

（3）将串口打印信息打开，串口打印信息设置：通过 adb shell 命令进入目录/proc/sys/kernel，输入 echo 8 4 1 7 > printk 将打印等级设为 8。gsensor 驱动中所有的打印信息打开，查看驱动程序的配置信

息读取状态以及 I2C 的初始化状态。

（4）查看 i2c 通信状态，当串口打印信息显示 i2C 通信失败时，主要有以下两个原因：

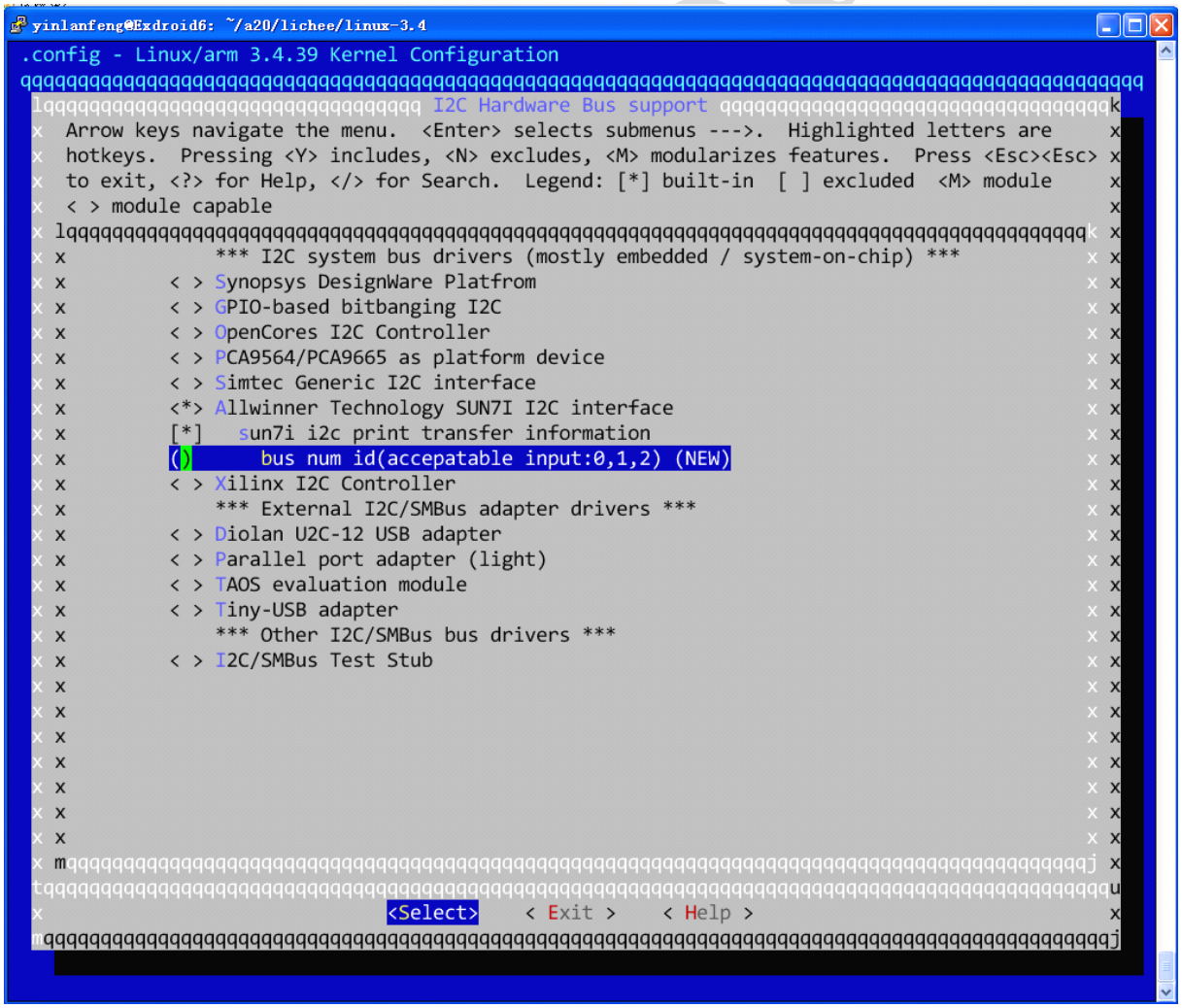
一是硬件上的，各个信号线接触不良，所以出现通信失败时，检查各引脚接触情况和电压情况。

二是因为 I2C 的地址不正确导致，因为 i2C 地址为 7 为地址，所以可能是因为在配置的时候没有移位或者是触控 IC 有多个 I2C 地址，导致地址不匹配。

若排除了以上两种原因后，还是不能正确进行 i2c 传输，则打开 i2c 传输打印，查看传输打印状态， 在编译服务器上，目录为 workspace\exdroid\lichee\linux-3.4 上，输入命令：

```
make ARCH=arm menuconfig
```

选 择 Device Drivers->I2C support->I2C Hardware Bus support -> sun7i_i2c_print_transfer_information，输入 Y 进入 bus num id(accepatable input:0,1,2,3,4)(NEW),输入数值,，若希望打印信息，数值对应相应的 I2C 号，gsensorIC 用的是第二组，因此选择数值为 2，若不希望打印信息，输入 N 退出保存即可。进行修改后，需要重新编译打包之后才能生效。



（5）查看 probe 是否成功，如 probe 不成功，根据打印信息定位驱动的运行情况，是因为什么

原因导致失败。

（6）当 probe 成功之后，旋转机器，是否看到界面旋转。

不能看到界面旋转的效果，问题确认步骤：

一、使用 adb shell getevent 命令，查看是否有 gsensor 坐标上报。

① 有数据上报

旋转机器，看数据是否有明显的变化。

- 有明显变化

  请查看 hal 层是否接收到数据以及数据的 处理是否正确。

- 无明显变化

  请确认设备是否处于正常工作的状态。

② 无数据上报

请看步骤二

二、使用 getevent 命令没有坐标值上报。

在 shell 中进入目录：sys/class/input/inputX，X 为设备的编号。查看 enable 是否为 1，如下所示：

```
root@android:/sys/class/input/input3 # cat enable
cat enable
0
```

若为 0，请将其置 1 后使用 getevent 命令查看是否有数据。

```
root@android:/sys/class/input/input3 # echo 1 > enable
echo 1 > enable
```

① 有数据

说明 hal 层无法将使能写入，请确认接口是否写正确。

② 无数据

将驱动中数据打印打开，查看是否有数据打印，如果没有，请确认是否驱动存在问题，若有，请确认数据是否有变化。

（7）可使用 adb 工具查看驱动是否加载以及 gsensor 是否有反应，adb 工具需要安装设备对应的驱动。

使用 adb shell 工具查看驱动是否存在于机器中以及驱动是否已经加载，以及 gsensor 之后是否有反应。同时可以作为简单的调试工具，修改好的驱动 PUSH 到机器中，重启系统之后即可运行新的

驱动，不用重新打包（配置文件内容除外）。使用到的命令如下所示：

adb shell　登录设备的 shell

adb push xx.ko /vendor/modules　将触摸驱动通过 adb 工具 push 到机器中

cd /vendor/modules　进入 modules 目录

ls *.ko　查看机器中已经有了那些驱动

insmod　加载驱动模块

lsmod　查看系统中已经加载了那些模块

rmmod　卸载驱动

getevent　查看系统中已经注册了那些 input 设备及这些设备上报的数据

（8）使用 adb 工具时，push 失败，是因为文件没有可写的权限，将系统设置为可写即可。

Android 处于安全性考虑，将 system，root 文件系统挂接为只读文件系统，开发人员在开发过程中，可以重新 remount 成读写文件系统

运行时

adb shell

mount -o remount , rw /dev/block/nandc /system

mount -o remount , rw /dev/root/

编译时修改：

Init.sun7i.rc 2 句话

mount ext4　/dev/block/nandc /system ro remount

mount rootfs rootfs /ro remount

（9） 如何判断 gsensor 上报数据的偏差。

将 gsensor 芯片水平放置，观察它上报上来的 x y z 三个轴的数据，此时准确的数据应该为 x 轴为 0，y 轴为 0，z 轴为 1g。

7.3. G-sensor 测试

完成驱动及 android hal 层的移植调试后，就可以进行测试这款 G-sensor 的效果了。

首先，旋转屏幕，看是否可以进行正常的横竖屏切换以验证 G-sensor 的基本功能。如果横竖屏切换可以正常切换，但响应慢，可以提高轮询时间来解决。

其次，可以使用 sensorlist.apk 来测试设备上报的数据是否存在问题。

然后，可以下载一些游戏进行效果测试，如赛车类，RacingMoto73.o 等。

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.

