本文主要是以重力传感器作为例子进行解说的，和MTK的驱动架构稍微有点区别，但是大体差不多，代码分析的比较详细，适合Sensor驱动的初学者。

重力传感器驱动的功能，主要是向HAL层提供IOCTRL接口，并通过input设备上报数据。芯片实际数据的读取是采用i2c协议读取原始数据，并且作为i2c设备挂载在系统上工作的。

1、调用关系

    采用模块化的编程方式，以下介绍函数的调用关系。

module\_init(aac\_MMAxxxxx\_init);

module\_exit(aac\_MMAxxxxx\_exit);

模块中定义了驱动初始化和退出函数，具体实现如下

static int \_\_init aac\_MMAxxxxxFC\_init(void)

{

    int ret;

    if ((ret = i2c\_add\_driver(&aac\_MMAxxxxxFC\_i2c\_driver))) {

        printk(KERN\_WARNING "aac\_MMAxxxxxFC\_init failed. /n");

        return ret;

    }

    return ret;

}

static void  \_\_exit aac\_MMAxxxxxFC\_exit(void)

{

    i2c\_del\_driver(&aac\_MMAxxxxxFC\_i2c\_driver);

}

    调用i2c\_add\_driver函数将aac\_MMAxxxxxFC\_i2c\_driver驱动添加实现了初始化函数，exit函数则调用i2c\_del\_driver函数将aac\_MMAxxxxxFC\_i2c\_driver驱动删除。

    对于aac\_MMAxxxxxFC\_i2c\_driver驱动结构体，由6个参数实现该结构体。Driver用模块名来填充，probe、remove、suspend、resume分别用相对应的函数来填充。Id\_table用aac\_ MMAxxxxxFC\_id来填充索引表。实现代码如下：

static struct i2c\_driver aac\_MMAxxxxxFC\_i2c\_driver = {

    .driver = {

        .name = MMAxxxxx\_MODULE\_NAME,

    },

    .probe = aac\_MMAxxxxxFC\_probe,

    .remove = aac\_MMAxxxxxFC\_remove,

    .id\_table   = aac\_MMAxxxxxFC\_id,

#ifndef CONFIG\_HAS\_EARLYSUSPEND

    .suspend  = mmaxxxxx\_suspend,

    .resume   = mmaxxxxx\_resume,

#endif

};

1.1、   对于结构体中索引表aac\_MMAxxxxxFC\_id数组，具体实现如下：

static const struct i2c\_device\_id aac\_MMAxxxxxFC\_id[] = {

    { MMAxxxxx\_MODULE\_NAME, 0 },

    { }

};

1.2、   MMAxxxxx\_MODULE\_NAME表示驱动名。在头文件中定义了具体名字"mmaxxxxx"。

1.3、   aac\_MMAxxxxxFC\_probe函数是i2c驱动寻找设备的经典实现，这里将具体分析下实现过程。实现思路是首先注册i2c功能函数类型，然后分配misc设备空间并注册，接下来分配输入设备空间并注册，注意将misc设备获取数据传给input设备数据中。最后创建工作队列，实现位置信息数据处理。

具体代码如下：

1.3.1调用i2c\_check\_functionality，函数返回我们需要类型的i2c适配器

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

        ret = -ENODEV;

        goto exit0;

    }

并且将入口函数参数client赋值给静态全局变量g\_client

    g\_client = client;

1.3.2调用kzalloc函数给空结构体赋值，空结构体的意义在于寻址。调用sysfs\_create\_group函数将设备体创建到mmaxxxxx\_attr\_group组，主要为调试使用，关于具体调试文件系统将在文章后面章节介绍。调用misc\_register函数注册misc设备mmaxxxxx\_misc\_device。

mmaxxxxx\_misc\_data = kzalloc(sizeof(struct mmaxxxxx\_data), GFP\_KERNEL);

    if (!mmaxxxxx\_misc\_data) {

        ret = -ENOMEM;

        goto exit1;

    }

    //init sysfs entry

    ret = sysfs\_create\_group(&client->dev.kobj, &mmaxxxxx\_attr\_group);

    if (ret)

        goto exit2;

    //misc\_register

     ret = misc\_register(&mmaxxxxx\_misc\_device);

    if (ret < 0) {

         dev\_err(&client->dev, "mmaxxxxx\_device register failed/n");

         goto exit3;

        }

1.3.3调用input\_allocate\_device函数给input设备分配空间

        //input\_allocate

        input = input\_allocate\_device();

    if (!input) {

            ret = -ENOMEM;

            printk("input device allocate failed/n");

            goto exit4;

    }

对input各个属性项目填充，name、phys表示映射的物理端口、id.bustype、id.vendor、id.product、

    input->name = "mmaxxxxx";

    input->phys = "mmaxxxxx/input0";

    input->id.bustype = BUS\_HOST;

    input->id.vendor = 0x0001;

    input->id.product = 0x0001;

    input->id.version = 0x0100;

    //evbit选择了事件类型，absbit表示了绝对值的数据

    input->evbit[0] = BIT(EV\_ABS);

    input->absbit[0] = BIT(ABS\_X) | BIT(ABS\_Y) | BIT(ABS\_Z);

    //将input赋值给全局变量g\_input\_dev设备。

    g\_input\_dev = input;

    ret = input\_register\_device(g\_input\_dev);

    if (ret) {

            printk("unable to register input polled device /n");

            goto exit5;

    }

    //把misc设备数据赋值给输入设备系统数据

    input\_set\_drvdata(g\_input\_dev, mmaxxxxx\_misc\_data);

1.3.4调用init\_MUTEX函数将信号量申请为互斥体信号，在分别在mmaxxxxx\_misc\_ioctl、asensor\_thread、mmaxxxxx\_resume函数中调用，确保时间能够安全赋值。

然后创建工作队列asensor\_wq，调用INIT\_DELAYED\_WORK函数asensor\_thread数据处理函数添加到工作任务asensor\_delayed\_work当中。注意，此时工作队列并没有开始工作，需要mmaxxxxx\_misc\_ioctl接到HAL层的控制信息时才打开端口，进行工作。

    init\_MUTEX(&sem\_thread);

    asensor\_wq = create\_workqueue("MMAxxxxxFC\_workqueue");

    if (!asensor\_wq) {

        printk("can't create a workqueue/n");

        ret = -1;

        goto exit6;

    }

    INIT\_DELAYED\_WORK(&asensor\_delayed\_work, asensor\_thread);

1.4、   aac\_MMAxxxxxFC\_remove函数是作为probe函数的反过程实现的，主要是取消工作队列和释放相关资源。

具体代码实现如下

    ret = cancel\_delayed\_work(&asensor\_delayed\_work);

    if(ret == 0){

        flush\_workqueue(asensor\_wq);

    }

    destroy\_workqueue(asensor\_wq);

    asensor\_wq = NULL;

#ifdef CONFIG\_HAS\_EARLYSUSPEND

    unregister\_early\_suspend(&early\_suspend);

#endif

    //clean input

    mmaxxxxx\_input\_cleanup();

    //sysfs

    sysfs\_remove\_group(&client->dev.kobj, &mmaxxxxx\_attr\_group);

    //misc

    misc\_deregister(&mmaxxxxx\_misc\_device);

1.5、   mmaxxxxx\_suspend和mmaxxxxx\_resume函数主要是睡眠和唤醒时作用，关掉外设电源并释放占用的相关资源。唤醒时实现反过程。

static int mmaxxxxx\_suspend(struct i2c\_client \*ic, pm\_message\_t mesg)

{

    printk("########mmaxxxxx\_suspend");

    cancel\_delayed\_work\_sync(&asensor\_delayed\_work);

    aac\_MMAxxxxxFC\_close\_client();

    return 0;

}

static int mmaxxxxx\_resume(struct i2c\_client \*ic)

{

    printk("########mmaxxxxx\_resume");

    aac\_MMAxxxxxFC\_init\_client();

    down\_interruptible(&sem\_thread);

    time\_delay.tv\_sec = 0;

    time\_delay.tv\_usec = read\_interval;

    up(&sem\_thread);

    if(1 == thread\_flag)

    {

        printk("########resume!/n");

            queue\_delayed\_work(asensor\_wq, &asensor\_delayed\_work, timeval\_to\_jiffies(&time\_delay));

    }

    return 0;

}

2.     特殊处理函数

2.1 misc控制函数

主要作用是处理HAL层的IOCTL命令，起到打开、关闭的任务。

首先定义了混杂设备结构体mmaxxxxx\_misc\_device，该结构体体由3个field组成，第一个表示misc设备的此设备号，第二个为misc设备的名字，第三个为misc操作结构体。操作结构体由我们自行定义。

static struct miscdevice mmaxxxxx\_misc\_device = {

    .minor = MISC\_DYNAMIC\_MINOR,

    .name = "mmaxxxxx",

    .fops = &mmaxxxxx\_misc\_fops,

};

    然后定义Misc操作结构体，该结构体由3个field组成，第一个表示所有者，属性固定为本模块，即THIS\_MODULE。第二个表示打开函数，处理数据信息，第三个表示控制函数，处理misc设备的相关控制命令。

    static const struct file\_operations mmaxxxxx\_misc\_fops = {

    .owner = THIS\_MODULE,

    .open =  mmaxxxxx\_misc\_open,

    .ioctl = mmaxxxxx\_misc\_ioctl,

};

    作为传感器输入设备，打开函数使用的也是数据流，所以定位数据没有意义。这种情况下，不能简单不声明lseek操作，因为默认方法是允许定位的。默认定位的方法是调用lseek函数在数据区往上或往下定位数据。在open方法中调用nonseekable\_open()时，它会通知内核设备不支持lseek。

    函数实现如下：

    static int mmaxxxxx\_misc\_open(struct inode \*inode, struct file \*file)

{

    int err;

    err = nonseekable\_open(inode, file);

    if (err < 0)

        return err;

    file->private\_data = mmaxxxxx\_misc\_data;

    return 0;

}

这里注意的是文件的私有数据赋值对象为mmaxxxxx\_misc\_data，是一个空结构体变量。难道也仅仅是为了寻址么？

    Ioctl函数作为misc设备核心的操作函数，主要作用是通过HAL层中相关command字的控制，给应用层提供了控制方法，最终实现设备体的状态获取，延时，激活，关闭，如匹配字不符合，则控制参数有误退出。

   Ioctl函数中主要包括的控制命令为MMAxxxxx\_IOCTL\_GET\_STATE、MMAxxxxx\_IOCTL\_SET\_DELAY、MMAxxxxx\_IOCTL\_ENABLE、MMAxxxxx\_IOCTL\_DISABLE几个命令。

具体可参考代码。如MMAxxxxx\_IOCTL\_GET\_STATE中主要通过copy\_to\_user将线程标示位赋值给参数argp，从而获取状态。其他几个具体参考代码。

    static int mmaxxxxx\_misc\_ioctl(struct inode \*inode, struct file \*file,unsigned int cmd, unsigned long arg)

{

    void \_\_user \*argp = (void \_\_user \*)arg;

    int32\_t interval;

    switch( cmd )

    {

        case MMAxxxxx\_IOCTL\_GET\_STATE:

        {

            if(copy\_to\_user(argp, &thread\_flag, sizeof(thread\_flag)))

                return -EFAULT;

            //printk("MMAxxxxx\_IOCTL\_GET\_STAT/n");

            break;

        }

        case MMAxxxxx\_IOCTL\_ENABLE:

        {

            aac\_MMAxxxxxFC\_init\_client();

            thread\_flag = 1;

                down\_interruptible(&sem\_thread);

                time\_delay.tv\_sec = 0;

                time\_delay.tv\_usec = read\_interval;

                up(&sem\_thread);

                queue\_delayed\_work(asensor\_wq, &asensor\_delayed\_work, timeval\_to\_jiffies(&time\_delay));

                //printk("MMAxxxxx\_IOCTL\_ENABLE/n");

            break;

        }

    }

}

    在MMAxxxxx\_IOCTL\_ENABLE控制命令下，通过原子操作定义了延时的时间，将工作任务asensor\_delayed\_work添加到工作队列asensor\_wq中，这样就循环开始了该工作。

    2.2 工作函数asensor\_thread

    工作函数中主要是通过i2c线读取相关的输出数据。I2c读取的方式这里不再详述，这里主要通过调用i2c\_smbus\_read\_i2c\_block\_data函数，读取连续三个地址的数值，通过数据处理，根据硬件相关的贴片方式，输出正确的xyz结果。

    处理过的结果用自定义的结构体保存。

struct \_mmaxxxxx\_data{

    int  x\_data;

    int  y\_data;

    int  z\_data;

};

    这里有个需要处理的地方就是有些芯片灵敏度过高，可以通过滤波算法进行相关的去抖动处理。具体参考后续文章。

3   调试信息控制文件接口

    这里通过static int g\_print = 0来实现是否输出打印信息，介绍相关知识之前，需要先了解linux内核中sys文件系统的介绍。sysfs 属性的功能只能靠阅读源代码来理解。在内核中， sysfs 属性一般是由 \_\_ATTR 系列的宏来声明的，如对设备的使用 DEVICE\_ATTR ，对总线使用 BUS\_ATTR ，对驱动使用 DRIVER\_ATTR ，对类别(class)使用CLASS\_ATTR, 这四个高级的宏来自于 <include/linux/device.h>, 都是以更低层的来自 <include/linux/sysfs.h> 中的\_\_ATTR/\_\_ATRR\_RO 宏实现； 因此我们在内核源码树中相应位置 drivers/scsi/ 找到这几个宏的使用情况，可以得到在drivers/scsi/scsi\_sysfs.c 中。

      下面通过代码介绍DEVICE\_ATTR的添加过程。

      3.1定义控制变量

static int g\_print = 0;

    3.2定义mmaxxxxx\_show\_print函数

功能主要是将g\_print打印到内存当中。

static ssize\_t mmaxxxxx\_show\_print(struct device \*dev,struct device\_attribute \*attr, char \*buf)

{

    return sprintf(buf, "%d/n", g\_print);

}

    3.3定义mmaxxxxx\_store\_print函数

    功能主要是获取buf中存在的控制值。

static ssize\_t mmaxxxxx\_store\_print(struct device \*dev,struct device\_attribute \*attr, char \*buf,size\_t count)

{

    unsigned long val = simple\_strtoul(buf, NULL, 10);

    //adjust\_light(val);

        //ggg=val;

    g\_print = val;

    return count;

}

    3.4填充设备属性DEVICE\_ATTR

static DEVICE\_ATTR(print, S\_IWUSR | S\_IRUGO,mmaxxxxx\_show\_print, mmaxxxxx\_store\_print);

其中有四个参数，分别表示是称、权限位、读函数、写函数。

      3.5属性数组mmaxxxxx\_attributes

主要是填充设备属性位置。

      static struct attribute \*mmaxxxxx\_attributes[] = {

      &dev\_attr\_print.attr,

      NULL

};

      3.6将属性数组加到属性组（group）里。

static const struct attribute\_group mmaxxxxx\_attr\_group = {

      .attrs = mmaxxxxx\_attributes,

};

至此完成了属性组的添加工作，通过adb连接去硬件系统中对应的文件为sys/devices/i2c-0/x-xxxx/print，x-xxxx对应的是芯片的地址线。

4    总结

文章中采用标准模块化的方法，调用内核函数，将i2c模块挂载到内核系统当中，并通过misc设备留接口给上层提供调用。在模块工作过程中，通过i2c读函数获取了实时的位置信息，并通过input设备将数据上报给用户层。