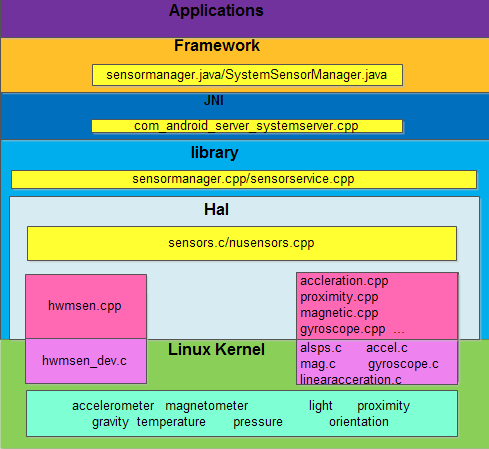
Sensor系统流程之kernle/hal层（mtk版）

Sensor架构可参见”driver\_all\_in\_one”描述。下图是以代码组织的Diagram。



1. Kernel层

目前Mtk自己实现的和原生的并存，这里介绍mtk自己实现的，也是现在很多平台上用的（hal层及以上都是原生的，差异只体现在kernel层）。

源码路径：

aosp/kernel-3.10/drivers/misc/mediateck/<sensor-driver>/xxx

aosp/kernel-3.10/drivers/misc/mediateck/hwmon/xxx

aosp/kernel-3.10/drivers/misc/mediateck/mach/<platform>/<project>/<sensor-driver>/cust\_acc.c

* **驱动注册**

**hwmsen\_dev.c**类似于interface，作为沟通hal层与sensor设备驱动间的桥梁，统一管理各种sensor驱动。这里mtk只提供了alsps/gsensor/msensor三种sensor，但可以自己添加。驱动填充struct sensor\_init\_info对象然后调用hwmsen\_alsps\_sensor\_add()(以光感为例，类似有hwmsen\_gsensor\_sensor\_add（）)添加到hwmsen\_dev.c定义好的struct sensor\_init\_info结构数组内（如static struct sensor\_init\_info\* alsps\_init\_list[MAX\_CHOOSE\_G\_NUM]），然后由hwmsen\_dev.c调用alsps\_sensor\_probe（）统一注册sensor驱动。

struct sensor\_init\_info定义如下：

struct sensor\_init\_info {

char \*name; //sensor设备的名字

int (\*init) (void); // alsps\_sensor\_probe（）调用，注册sensor-driver

int (\*uninit) (void); //对应init的析构函数

struct platform\_driver \*platform\_diver\_addr; //不要定义，hwmsen\_dev.c统一添加

};

以下以ltr559.c举例：

定义sturct sensor\_init\_info对象ltr559\_init\_info，挂载.init/.uninit函数

static struct sensor\_init\_info *ltr559\_init\_info* = {

.name = "ltr559",

.init = ltr559\_local\_init,

.uninit = ltr559\_remove,

};

ltr559\_local\_init()调用i2c\_add\_driver（）注册i2c驱动。 ltr559.c在模块加载（module\_init）时执行ltr559\_init()调用hwmsen\_alsps\_sensor\_add ()将定义的ltr559\_init\_info对象添加到hwmsen\_dev.c预先定义好的结构体指针数组alsps\_init\_list内。

int

hwmsen\_alsps\_sensor\_add(struct sensor\_init\_info\* obj)

{

int err=0;

int i =0;

for (i =0; i < MAX\_CHOOSE\_G\_NUM; i++ ) {

if (NULL == alsps\_init\_list[i]) {

alsps\_init\_list[i]=kzalloc(sizeof(structsensor\_init\_info),

GFP\_KERNEL);

obj->platform\_diver\_addr = &alsps\_sensor\_driver;

alsps\_init\_list[i] = obj; //添加ltr559\_init\_info对象

break;

}

}

return err;

}

hwmsen\_init（）调用platform\_driver\_register（）注册platform驱动。其probe函数alsps\_sensor\_probe（）调用ltr559\_init\_info对象的.init注册i2c驱动

static int

alsps\_sensor\_probe(struct platform\_device \*pdev)

{

int i =0;

for (i = 0; i < MAX\_CHOOSE\_G\_NUM; i++) {

if (NULL != alsps\_init\_list[i]) {

alsps\_init\_list[i]->init(); //调用ltr559\_local\_init（）注册i2c驱动

}

}

return 0;

}

至此驱动注册完成

* **控制流程**

代码的操作对象是数据。这里先介绍kernel层用到的重要数据结构。hwmsen\_dev.c定义sensor对象类型是struct hwmdev\_object，如下：

struct hwmdev\_object {

struct input\_dev \*idev; //输入设备，sensor有数据更新时报告事件

struct miscdevice mdev; //与hal层交互，控制sensor操作（enable/delay/get-data）

struct dev\_context \*dc; //桥梁，代表多个sensor设备

struct work\_struct report; //与下面的定时器一起，当enable-sensor时主动读取sensor数据，放在全局的对象struct hwmsen\_data obj\_data内

struct timer\_list timer; /\* polling timer \*/

atomic\_t delay; /\*polling period for reporting input event\*/

atomic\_t wake; /\*user-space request to wake-up, used with stop\*/

atomic\_t trace;

uint32\_t active\_sensor; // Active, but hwmsen don't need data sensor. Maybe other need it's data.

uint32\_t active\_data\_sensor; // Active and hwmsen need data sensor.

#if defined(CONFIG\_HAS\_EARLYSUSPEND)

struct early\_suspend early\_drv;

#ifdef CONFIG\_PM\_WAKELOCKS

struct wakeup\_source read\_data\_wake\_lock;

#else

struct wake\_lock read\_data\_wake\_lock;

#endif

atomic\_t early\_suspend;

//add for fix resume end

#endif

};

hwmsen\_dev.c模块看到的sensor设备类型准确说是struct hwmsen\_context类型，包括struct hwmsen\_object（ltr559.c设备对象，定义sensor工作模式与控制操作的方法）。仿c++的内部类，定义如下：

struct hwmsen\_context { //sensor设备

atomic\_t enable;

atomic\_t delay;

uint32\_t delayCountSet;

uint32\_t delayCount;

struct hwmsen\_object {

void \*self;

int polling; //工作模式0-中断 1-轮询

/\*hwmsen\_dev.c模块对sensor的操作方法包括enable/delay/get-data\*/

int (\*sensor\_operate) (void \*self, uint32\_t command,

void \*buff\_in, int size\_in,

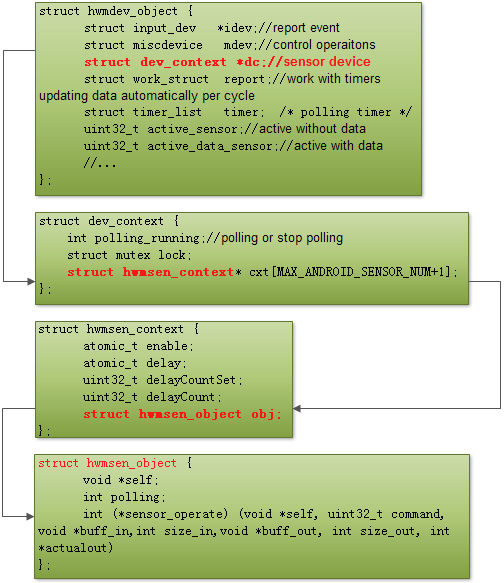
void \*buff\_out, int size\_out, int \*actualout);

};

struct hwmsen\_object obj;

};

下图描绘了这几个重要数据结构间的关系：



Ltr559.c)

Hwmsen\_dev.c

可以说struct hwmdev\_object对象代表了kernel层sensor，hal层与kernel层打交道就是围绕这个对象操作的。

hwmsen\_dev.c声明了struct hwmdev\_object类型的全局变量:

static struct hwmdev\_object \*hwm\_obj = NULL;

hwmsen\_probe()初始化hwm\_obj对象：

static int

hwmsen\_probe(struct platform\_device \*pdev)

{

//初始化定时器及work\_struct，定时时间到周期性调度work\_struct获取sensor数据

hwm\_obj = hwmsen\_alloc\_object();

//分配并注册输入设备，当有新数据获得时报告输入事件

hwm\_obj->idev = input\_allocate\_device();

//...

if((err = input\_register\_device(hwm\_obj->idev))) {

goto exit\_input\_register\_device\_failed;

}

//注册miscdev，用于hal层通过ioctl控制sensor的enable/delay/get-data

hwm\_obj->mdev.minor = MISC\_DYNAMIC\_MINOR;

hwm\_obj->mdev.name = HWM\_SENSOR\_DEV\_NAME;

hwm\_obj->mdev.fops = &hwmsen\_fops;

if((err = misc\_register(&hwm\_obj->mdev))) {

goto exit\_misc\_register\_failed;

}

return err;

}

mtk自己实现的kernel通用模块hwmsen\_dev.c与hal层是通过ioctl方式控制sensor工作的，如下：

static long

hwmsen\_unlocked\_ioctl(struct file \*fp, unsigned int cmd, unsigned long arg)

{

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

uint32\_t flag;

struct sensor\_delay delayPara;

hwm\_trans\_data hwm\_sensors\_data;

int i = 0;

atomic\_t delaytemp;

atomic\_set(&delaytemp, 200);//used to finding fastest sensor polling rate

switch (cmd) {

case HWM\_IO\_SET\_DELAY://设置polling时间

// android2.3 sensor system has 4 sample delay 0ms 20ms 60ms 200ms

if (copy\_from\_user(&delayPara, argp, sizeof(delayPara))) {

return -EFAULT;

}

hwmsen\_set\_delay(delayPara.delay,delayPara.handle);//modified for android2.3

update\_workqueue\_polling\_rate(delayPara.delay);

break;

case HWM\_IO\_SET\_WAKE://hal层唤醒sensor，报同步事件

hwmsen\_wakeup(hwm\_obj);

break;

case HWM\_IO\_ENABLE\_SENSOR://使能sensor，激活定时器，轮询sensor数据

if(copy\_from\_user(&flag, argp, sizeof(flag)))

{

return -EFAULT;

}

hwmsen\_enable(hwm\_obj, flag, 1);

break;

case HWM\_IO\_DISABLE\_SENSOR://关闭sensor，停止定时器

if(copy\_from\_user(&flag, argp, sizeof(flag)))

{

return -EFAULT;

}

hwmsen\_enable(hwm\_obj, flag, 0);

break;

/\*

\*获取sensor数据，轮询的sensor数据都放在obj\_data内保存

\*（obj\_data这里担当数据池，存储各个sensor获得的数据）

\*hal层直接从此拿去数据

\*/

case HWM\_IO\_GET\_SENSORS\_DATA:

if(copy\_from\_user(&hwm\_sensors\_data, argp, sizeof(hwm\_sensors\_data)))

{

return -EFAULT;

}

mutex\_lock(&obj\_data.lock);

memcpy(hwm\_sensors\_data.data, &(obj\_data.sensors\_data),

sizeof(hwm\_sensor\_data) \* MAX\_ANDROID\_SENSOR\_NUM);

for(i = 0; i < MAX\_ANDROID\_SENSOR\_NUM; i++) {

//data\_type indicates which sensor it is

if(hwm\_sensors\_data.date\_type & 1<<i){

hwm\_sensors\_data.data[i].update = 1;

} else{

hwm\_sensors\_data.data[i].update = 0;

}

}

mutex\_unlock(&obj\_data.lock);

if(copy\_to\_user(argp, &hwm\_sensors\_data, sizeof(hwm\_sensors\_data)))

{

return -EFAULT;

}

break;

//使能sensor但不会激活定时器，不会轮询sensor数据

case HWM\_IO\_ENABLE\_SENSOR\_NODATA:

if(copy\_from\_user(&flag, argp, sizeof(flag)))

{

return -EFAULT;

}

hwmsen\_enable\_nodata(hwm\_obj, flag, 1);

break;

case HWM\_IO\_DISABLE\_SENSOR\_NODATA://关闭sensor

if(copy\_from\_user(&flag, argp, sizeof(flag)))

{

return -EFAULT;

}

hwmsen\_enable\_nodata(hwm\_obj, flag, 0);

break;

default:

return -ENOIOCTLCMD;

}

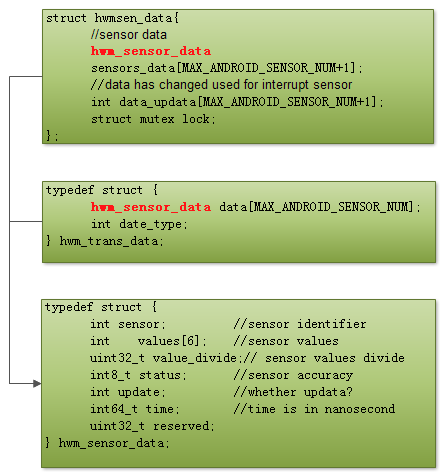
return 0;

}

上面提到轮询的sensor数据是预先放在数据池obj\_data内，ojb\_data定义如下：

static struct hwmsen\_data obj\_data;

struct hwmsen\_data描述kernel层sensor数据类型,下图是有关sensor数据的类型的定义：



在使能时会激活定时器，周期性调度hwmsen\_work\_func()读取sensor数据。

static int

hwmsen\_enable(struct hwmdev\_object \*obj, int sensor, int enable)

{

struct hwmsen\_context \*cxt = NULL;

int err = 0;

uint32\_t sensor\_type;

sensor\_type = 1 << sensor;

mutex\_lock(&obj->dc->lock);

cxt = obj->dc->cxt[sensor];//获取注册的sensor（类型struct hwmsen\_context）

if(enable == 1) { //使能

enable\_again = true;

obj->active\_data\_sensor |= sensor\_type;//激活数据sensor

if((obj->active\_sensor & sensor\_type) == 0)

{

if(cxt->obj.sensor\_operate(cxt->obj.self,

SENSOR\_ENABLE, &enable,sizeof(int), NULL, 0, NULL) != 0){}

update\_workqueue\_polling\_rate(200);

atomic\_set(&cxt->enable, 1);

}

if((0 == obj->dc->polling\_running) && (obj->active\_data\_sensor != 0))

{

obj->dc->polling\_running = 1;

mod\_timer(&obj->timer,

jiffies + atomic\_read(&obj->delay)/(1000/HZ)); //激活定时器调度work\_struct更新sensor数据

}

} else if ((enable == 0)) {

obj->active\_data\_sensor &= ~sensor\_type;

if((obj->active\_sensor & sensor\_type) == 0) // no no-data active

{

if(cxt->obj.sensor\_operate(cxt->obj.self, SENSOR\_ENABLE, &enable,sizeof(int), NULL, 0, NULL) != 0)

{

HWM\_ERR("deactiva sensor(%d) err = %d\n", sensor, err);

err = -EINVAL;

goto exit;

}

atomic\_set(&cxt->enable, 0);

update\_workqueue\_polling\_rate(200);// re-update workqueue polling rate

}

if((1 == obj->dc->polling\_running) && (obj->active\_data\_sensor == 0))

{

obj->dc->polling\_running = 0;

del\_timer\_sync(&obj->timer);

cancel\_work\_sync(&obj->report);

}

obj\_data.sensors\_data[sensor].values[0] = SENSOR\_INVALID\_VALUE;

obj\_data.sensors\_data[sensor].values[1] = SENSOR\_INVALID\_VALUE;

obj\_data.sensors\_data[sensor].values[2] = SENSOR\_INVALID\_VALUE;

}

mutex\_unlock(&obj->dc->lock);

return err;

}

hwmsen\_poll（）周期性调度hwmsen\_work\_func()，获取sensor数据，放在obj\_data内并上报事件

static void

hwmsen\_work\_func(struct work\_struct \*work)

{

struct hwmdev\_object \*obj = hwm\_obj;

struct hwmsen\_context \*cxt = NULL;

int out\_size;

hwm\_sensor\_data sensor\_data;

uint32\_t event\_type = 0;

int64\_t nt;

struct timespec time;

int err, idx;

//获取所有sensor数据

for(idx = 0; idx < MAX\_ANDROID\_SENSOR\_NUM; idx++)

{

cxt = obj->dc->cxt[idx];

//在中断下工作的sensor，调用hwmsen\_get\_interrupt\_data()更新obj\_data

if(cxt->obj.polling == 0)

{

if(obj\_data.data\_updata[idx] == 1)

{

mutex\_lock(&obj\_data.lock);

event\_type |= (1 << idx);

obj\_data.data\_updata[idx] = 0;

mutex\_unlock(&obj\_data.lock);

}

//Even if interrupt mode, try to take the initiative in querying a valid sensor data.

else if (obj\_data.sensors\_data[idx].values[0]

!= SENSOR\_INVALID\_VALUE)

{

continue;

}

}

//读取轮询sensor数据

err = cxt->obj.sensor\_operate(cxt->obj.self,SENSOR\_GET\_DATA,

NULL, 0, &sensor\_data, sizeof(hwm\_sensor\_data), &out\_size);

if((idx == ID\_LIGHT) ||(idx == ID\_PRESSURE) ||(idx == ID\_PROXIMITY)

|| (idx == ID\_TEMPRERATURE))

{

// data changed, update the data

if(sensor\_data.values[0] != obj\_data.sensors\_data[idx].values[0])

{

//sensor数据放在obj\_data内

mutex\_lock(&obj\_data.lock);

obj\_data.sensors\_data[idx].values[0] = sensor\_data.values[0];

obj\_data.sensors\_data[idx].value\_divide

=sensor\_data.value\_divide;

obj\_data.sensors\_data[idx].status = sensor\_data.status;

obj\_data.sensors\_data[idx].time = nt;

event\_type |= (1 << idx);

mutex\_unlock(&obj\_data.lock);

}

}

else

{

// data changed, update the data

if((sensor\_data.values[0] != obj\_data.sensors\_data[idx].values[0])

|| (sensor\_data.values[1]

!= obj\_data.sensors\_data[idx].values[1])

|| (sensor\_data.values[2]

!= obj\_data.sensors\_data[idx].values[2])

|| (idx == ID\_MAGNETIC)

|| (idx == ID\_ACCELEROMETER)

|| (idx == ID\_GYROSCOPE)

|| (idx == ID\_ORIENTATION))

{

if((0 == sensor\_data.values[0]

&& 0==sensor\_data.values[1]

&& 0 == sensor\_data.values[2])

&& (idx != ID\_GYROSCOPE))

{

continue;

}

mutex\_lock(&obj\_data.lock);

obj\_data.sensors\_data[idx].values[0] = sensor\_data.values[0];

obj\_data.sensors\_data[idx].values[1] = sensor\_data.values[1];

obj\_data.sensors\_data[idx].values[2] = sensor\_data.values[2];

obj\_data.sensors\_data[idx].value\_divide=

sensor\_data.value\_divide;

obj\_data.sensors\_data[idx].status = sensor\_data.status;

obj\_data.sensors\_data[idx].time = nt;

event\_type |= (1 << idx);

mutex\_unlock(&obj\_data.lock);

}

}

}

if(event\_type != 0)

//报告事件有数据更新好了，通知hal层读取

input\_report\_rel(obj->idev, EVENT\_TYPE\_SENSOR, event\_type);

input\_sync(obj->idev);//modified

}

if(obj->dc->polling\_running == 1)

{

//sensor还在工作，继续下一轮读取

mod\_timer(&obj->timer,

jiffies + atomic\_read(&obj->delay)/(1000/HZ));

}

}

中断模式下工作的sensor在中断处理函数内调用hwmsen\_get\_interrupt\_data（）将sensor数据放到obj\_data内

int

hwmsen\_get\_interrupt\_data(int sensor, hwm\_sensor\_data \*data)

{

struct dev\_context \*mcxt = &dev\_cxt;

struct hwmdev\_object \*obj = hwm\_obj;

int64\_t nt;

if((sensor == ID\_LIGHT) ||(sensor == ID\_PRESSURE)

||(sensor == ID\_PROXIMITY) || (sensor == ID\_TEMPRERATURE))

{

// data changed, update the data

if(data->values[0] != obj\_data.sensors\_data[sensor].values[0])

{//将中断获得数据放到obj\_data内

mutex\_lock(&obj\_data.lock);

obj\_data.data\_updata[sensor] = 1;

obj\_data.sensors\_data[sensor].values[0] = data->values[0];

obj\_data.sensors\_data[sensor].time = nt;

obj\_data.sensors\_data[sensor].value\_divide = data->value\_divide;

mutex\_unlock(&obj\_data.lock);

}

}

if(obj->dc->polling\_running == 1)

{

hwmsen\_work\_func(NULL);//调用这个函数报告事件

}

return 0;

}

hwmsen\_dev.c对sensor的操作都是调用cxt->obj.sensor\_operate（）完成的。驱动必须提供初始化的struct hwmsen\_object对象。接着看下ltr559.c的ltr559\_i2c\_probe（）函数。

static int

ltr559\_i2c\_probe(struct i2c\_client \*client, const struct i2c\_device\_id \*id)

{

struct ltr559\_priv \*obj;

struct hwmsen\_object obj\_ps, obj\_als;

int err = 0;

//工厂模式调用接口

if(err = misc\_register(&ltr559\_device))

{

goto exit\_misc\_device\_register\_failed;

}

//初始化struct hwmsen\_object对象

//填充.self/.polling/.sensor\_operate成员

obj\_als.self = ltr559\_obj;

obj\_als.polling = 1;

obj\_als.sensor\_operate = ltr559\_als\_operate;

if(err = hwmsen\_attach(ID\_LIGHT, &obj\_als))//挂载到hwmsen\_dev.c定义的dev\_cxt对象内

{

goto exit\_create\_attr\_failed;

}

return err;

}

hwmsen\_attach（）部分代码如下：

int

hwmsen\_attach(int sensor, struct hwmsen\_object \*obj)

{

struct dev\_context \*mcxt = &dev\_cxt;

int err = 0;

mcxt->cxt[sensor] = kzalloc(sizeof(struct hwmsen\_context), GFP\_KERNEL);

if(mcxt->cxt[sensor] == NULL)

{

err = -EPERM;

goto err\_exit;

}

atomic\_set(&mcxt->cxt[sensor]->enable, 0);

memcpy(&mcxt->cxt[sensor]->obj, obj, sizeof(\*obj));//将定义好的hwmsen\_object对象挂载到dev\_cxt内；这样hwmsen\_dev.c便可操作sensor了

return err;

}

下面看下.sensor\_operate提供的操作方法：

int

ltr559\_als\_operate(void\* self, uint32\_t command, void\* buff\_in, int size\_in,

void\* buff\_out, int size\_out, int\* actualout)

{

int err = 0;

int value;

hwm\_sensor\_data\* sensor\_data;

struct ltr559\_priv \*obj = (struct ltr559\_priv \*)self;

switch (command){

case SENSOR\_DELAY://delay

if((buff\_in == NULL) || (size\_in < sizeof(int)))

{

err = -EINVAL;

}

// Do nothing

break;

case SENSOR\_ENABLE://enable/disable

if((buff\_in == NULL) || (size\_in < sizeof(int)))

{

err = -EINVAL;

} else {

value = \*(int \*)buff\_in;

if(value) {

err = ltr559\_als\_enable(als\_gainrange);

if(err < 0) {

return -1;

}

set\_bit(CMC\_BIT\_ALS, &obj->enable);

} else {

err = ltr559\_als\_disable();

if(err < 0){

return -1;

}

clear\_bit(CMC\_BIT\_ALS, &obj->enable);

}

}

break;

case SENSOR\_GET\_DATA://获取sensor数据

if((buff\_out == NULL) || (size\_out< sizeof(hwm\_sensor\_data)))

{

err = -EINVAL;

} else {

sensor\_data = (hwm\_sensor\_data \*)buff\_out;

obj->als = ltr559\_als\_read(als\_gainrange);

}

break;

default:

err = -1;

break;

}

return err;

}

* **工厂模式**

MTK legacy 并没有提供统一的工厂模式借口，由驱动自己实现，如ltr559\_i2c\_probe（）调用misc\_register（）注册工厂模式接口ioctl

static int

ltr559\_unlocked\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)

{

struct i2c\_client \*client = (struct i2c\_client\*)file->private\_data;

struct ltr559\_priv \*obj = i2c\_get\_clientdata(client);

int err = 0;

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

int dat;

uint32\_t enable;

switch (cmd){

case ALSPS\_SET\_PS\_MODE:

if(copy\_from\_user(&enable, ptr, sizeof(enable)))

{

err = -EFAULT;

goto err\_out;

}

if(enable)

{

err = ltr559\_ps\_enable(ps\_gainrange);

if(err < 0)

{

goto err\_out;

}

set\_bit(CMC\_BIT\_PS, &obj->enable);

}

else

{

err = ltr559\_ps\_disable();

if(err < 0)

{

goto err\_out;

}

clear\_bit(CMC\_BIT\_PS, &obj->enable);

}

break;

case ALSPS\_GET\_PS\_MODE:

enable = test\_bit(CMC\_BIT\_PS, &obj->enable) ? (1) : (0);

if(copy\_to\_user(ptr, &enable, sizeof(enable)))

{

err = -EFAULT;

goto err\_out;

}

break;

case ALSPS\_GET\_PS\_DATA:

obj->ps = ltr559\_ps\_read();

if(obj->ps < 0)

{

goto err\_out;

}

dat = ltr559\_get\_ps\_value(obj, obj->ps);

if(copy\_to\_user(ptr, &dat, sizeof(dat)))

{

err = -EFAULT;

goto err\_out;

}

break;

//…

}

1. Hal层

hal层源码都在路径aosp/vendor/mediatek/proprietary/hardware/sensor/xxx：

其中主要的接口文件如下：

aosp/vendor/mediatek/proprietary/hardware/sensor/sensors.c

aosp/vendor/mediatek/proprietary/hardware/sensor/nusensors.cpp

aosp/vendor/mediatek/proprietary/hardware/sensor/hwmsen.cpp（MTK自己实现的sensor）

aosp/vendor/mediatek/proprietary/hardware/sensor/proximity.cpp(android原生，每一个这样的文件都代表一个sensor)

// aosp/vendor/mediatek/proprietary/hardware/sensor/...

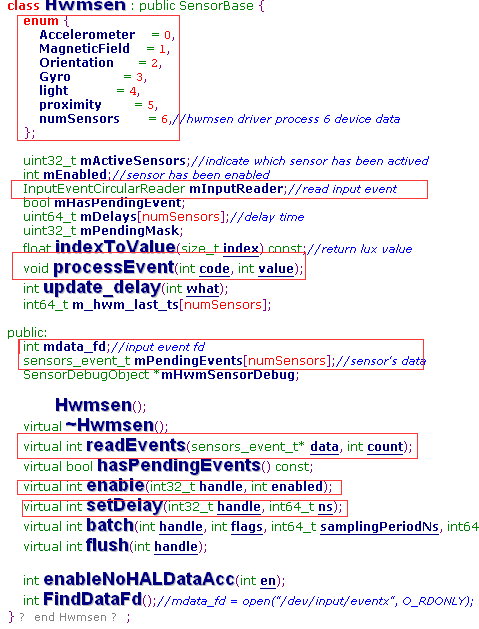
* **hwmsen.cpp**

hwmsen.cpp文件定义的class Hwmsen就是MTK自己的hal层sensor，描述一个sensor具有的属性与操作方法。基类class SensorBase描述了所有的sensor共同的属性，每一个hal层sensor都从SensorBase继承而来。SensorBase内定义了很多虚函数，基类的指针或引用可以指向派生类对象，从而实现动态特性，调用相应派生类的方法。

Sensor为抽象基类，不能实例化，必须被继承，其定义如下：



class hwmsen是SensorBase的派生类，其定义如下：



所有sensor共有的操作方法

构造函数Hwsen()调用基类SensorBase的构造函数，传递参数给dev\_name = “/dev/hwmsensor”; data\_name = “hwmdata”;初始化如mPendingEvents成员，调用FindDataFd()返回打开的文件/dev/input/eventx的文件描述符保存在mdata\_fd内，调用基类的open\_device()返回打开的文件/dev/hwmsensor的描述符保存在基类保护型成员dev\_fd内。这样hal层并可通过文件描述符dev\_fd通过ioctl控制hwmsen\_dev.c模块sensor 的使能、延迟、获得数据（如kernel层描述）。

下面结合代码看下sensor基本的操作enable/setDelay/read

int Hwmsen::

enable(int32\_t handle, int en)//handle代表使能某个sensor

{

int io\_value = 0;

int i=0;

int flags = en ? 1 : 0;

int err = 0;

uint32\_t sensor = (1 << handle); // new active/inactive sensor

if( handle != ID\_ACCELEROMETER && handle != ID\_MAGNETIC &&

handle != ID\_ORIENTATION && handle != ID\_GYROSCOPE &&

handle != ID\_PROXIMITY && handle != ID\_LIGHT)

{//目前mtk的hal sensor只支持这6个类型sensor

return 0;

}//not supported by hwmsen

if(en == 1)//en == 1 使能； en == 0 关闭

{

switch(handle)

{

case ID\_ACCELEROMETER:

m\_hwm\_last\_ts[Accelerometer] = 0;

break;

case ID\_MAGNETIC:

m\_hwm\_last\_ts[MagneticField] = 0;

break;

case ID\_ORIENTATION:

m\_hwm\_last\_ts[Orientation] = 0;

break;

case ID\_GYROSCOPE:

m\_hwm\_last\_ts[Gyro] = 0;

break;

case ID\_LIGHT:

m\_hwm\_last\_ts[light] = 0;

break;

case ID\_PROXIMITY:

m\_hwm\_last\_ts[proximity] = 0;

break;

}

//在构造函数已经获得dev\_fd文件描述符了，这里通过ioctl直接控制sensor使能

if(ioctl(dev\_fd, HWM\_IO\_ENABLE\_SENSOR, &handle))

{

return -1;

}

mEnabled = 1;

mActiveSensors |= sensor;

}

else

{//disable

mActiveSensors &= ~sensor;

if(0==mActiveSensors)

{

mEnabled = 0;

}

if(ioctl(dev\_fd, HWM\_IO\_DISABLE\_SENSOR, &handle))

{

return -1;

}

}

return err;

}

delay一方面设置sensor硬件的延迟需要sensor支持；另一面设置定时器polling的周期

int Hwmsen::

setDelay(int32\_t handle, int64\_t ns)

{

int what = -1;

if( handle != ID\_ACCELEROMETER && handle != ID\_MAGNETIC &&

handle != ID\_ORIENTATION && handle != ID\_GYROSCOPE &&

handle != ID\_LIGHT)

{

return 0;

}

what = handle;//指定某个sensor

if (uint32\_t(what) >= numSensors)

return -EINVAL;

if (ns < 0)

return -EINVAL;

mDelays[what] = ns;//纳秒

return update\_delay(what);//调用updata\_delay()

}

int Hwmsen::

update\_delay(int what)

{

struct sensor\_delay delayPara;

//if (mEnabled) //always update delay even sensor is not enabled.

{

delayPara.delay = mDelays[what]/1000000;//转为毫秒

delayPara.handle = what;

if(delayPara.delay < 10) //set max sampling rate = 100Hz

{

delayPara.delay = 10;

}

if(ioctl(dev\_fd, HWM\_IO\_SET\_DELAY, &delayPara))//设置delay

{

return -errno;

}

}

return 0;

}

readEvents()读取sensor数据，首先读取/dev/input/eventx看是否有事件更新，有的话才会调用processEvent()读取sensor数据保存在mPendingEvents内。

int Hwmsen::

readEvents(sensors\_event\_t\* data, int count)

{

int err=0;

int i=0;

if (count < 1) { return -EINVAL; }

ssize\_t n = mInputReader.fill(mdata\_fd);//读取/dev/input/eventx所有输入事件

if (n < 0) return n;

int numEventReceived = 0;

input\_event const\* event;

while (count && mInputReader.readEvent(&event)) {//获得mInputReader对象事件buffer内一个input\_event

int type = event->type;

if (type == EV\_REL){

processEvent(event->code, event->value);//有事件则调用processEvent()读取sensor数据放在mPendingEvents内

int64\_t time = android::elapsedRealtimeNano();//timevalToNano(event->time);

for (i=0 ; count && i<numSensors ; i++)

{

if (mActiveSensors && (1<<i)){

if (mPendingMask & (1<<i)){

mPendingMask &= ~(1<<i);

if(i<=3)

{//此处是校准获取事件的时间，省略

if(count != 0){

mPendingEvents[i].timestamp=time;

\*data++ = mPendingEvents[i];//sensor数据反馈给上层

count--;

numEventReceived++;

}

m\_hwm\_last\_ts[i] = mPendingEvents[i].timestamp;

}

else

{

if(count !=0)

{

\*data++ = mPendingEvents[i];

count--;

numEventReceived++;

}

}

}

}

}/end for

}

else if (type == EV\_SYN)

{

}

else

{

mInputReader.next();

}

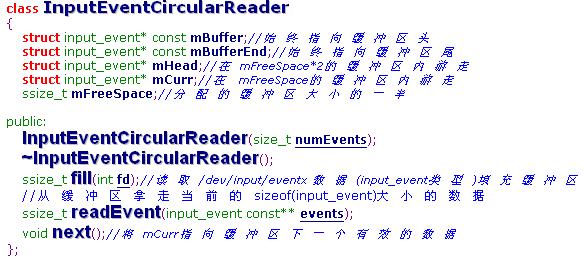
mInputReader.next();//处理下一个事件

}//end while

return numEventReceived;

}

Hwmsen对象有一个has-a关系的InputEventCircularReader mInputReader对象，负责读取输入事件。类型声明如下：



在构造函数Hwmsen（）已经分配好了缓冲区大小为4\*2\*sizeof(struct input\_event)字节大小(mBuffer = new input\_event[4\*2];)

fill()读取上报的事件

ssize\_t InputEventCircularReader::

fill(int fd)

{

size\_t numEventsRead = 0;

if (mFreeSpace) {

const ssize\_t nread = read(fd, mHead, mFreeSpace \* sizeof(input\_event));

if (nread<0 || nread % sizeof(input\_event)) {

return nread<0 ? -errno : -EINVAL;

}

/\*下面的算法将缓冲区分成两部分mBuffer始终指向缓冲区头;mBufferEend始

\*终指向缓冲区尾（分配buffer大小的一半）；mHead在缓冲区内游走，当读取

\*的数据超过mBufferEnd后则将数据拷贝到mBuffer指向的起始位置

\*/

numEventsRead = nread / sizeof(input\_event);

if (numEventsRead) {

mHead += numEventsRead;

mFreeSpace -= numEventsRead;

if (mHead > mBufferEnd) {

size\_t s = mHead - mBufferEnd;

memcpy(mBuffer, mBufferEnd, s \* sizeof(input\_event));

mHead = mBuffer + s;

}

}

}

return numEventsRead;

}

readEvent()从缓冲区读取sizoof(struct input\_event)字节的数据

ssize\_t InputEventCircularReader::

readEvent(input\_event const\*\* events)

{

\*events = mCurr; //mCurr类似于文件的offset，指向当前可用的input\_event

ssize\_t available = (mBufferEnd - mBuffer) - mFreeSpace;

return available ? 1 : 0;

}

next()更新mCurr指向下一个可用的input\_event

void InputEventCircularReader::

next()

{

mCurr++;

mFreeSpace++;

if (mCurr >= mBufferEnd) {

mCurr = mBuffer;

}

}

上面分析了辅助类InputEventCircularReader，下面回到Hwmsen看下processEvent()函数做的事情

void Hwmsen::

processEvent(int code, int value)

{

hwm\_trans\_data sensors\_data;//声明sensor\_data，保存获取的sensor数据

int err =0;

int i=0;

memset(&sensors\_data, 0 , sizeof(hwm\_trans\_data));

if(EVENT\_TYPE\_SENSOR != code) return;

sensors\_data.date\_type = value;//这里的value代表要读取数据的sensor

err = ioctl(dev\_fd, HWM\_IO\_GET\_SENSORS\_DATA, &sensors\_data);//读取sensor数据

for (i =0; i < MAX\_ANDROID\_SENSOR\_NUM; i++) {

if (sensors\_data.data[i].update == 0) {

continue;

}

switch (i) {

//既然能走到这就说明已经有数据更新了，对于根本不存在的sensor

//不必担心走不对应的case

case ID\_ORIENTATION:

//省略

break;

case ID\_MAGNETIC:

//…

break;

case ID\_ACCELEROMETER:

//…

break;

case ID\_GYROSCOPE:

break;

case ID\_PROXIMITY://近距数据处理，保存在mPendingEvents内

mPendingMask |= 1<<proximity;

mPendingEvents[proximity].type = SENSOR\_TYPE\_PROXIMITY;

mPendingEvents[proximity].sensor = sensors\_data.data[ID\_PROXIMITY].sensor;

mPendingEvents[proximity].distance = (float)sensors\_data.data[ID\_PROXIMITY].values[0];

mPendingEvents[proximity].timestamp = sensors\_data.data[ID\_PROXIMITY].time;

break;

case ID\_LIGHT:

break;

}

}

}

* **nusensors.cpp**

nusensors.cpp管理hal层所有的sensor。struct sensors\_poll\_context\_t类成员mSensors是结构体指针数组，数组每个元素指向SensorBase。mSensors包含了所有的sensor。

类sensors\_poll\_context\_t定义如下：

struct sensors\_poll\_context\_t {

/\*硬件抽象层模块，第一个成员必须是struct hw\_device\_t类型其必须位于声明

\*的类型开头

\* 关于硬件抽象层模块规则参见之前总结文档《android光照系统实现》

\*/

struct sensors\_poll\_device\_1 device;// must be first

/\*C++的类一种是以关键字class声明，另一种以关键字struct声明。以struct

\*声明的类型当有成员函数声明时则编译器视其为类的定义而不是简单的结构体

\*类型。struct关键字声明的类当不指定成员访问权限时，默认为public（与class

\*相反）

\*/

sensors\_poll\_context\_t();

~sensors\_poll\_context\_t();

int activate(int handle, int enabled);//激活sensor

int setDelay(int handle, int64\_t ns);//设置polling周期

int pollEvents(sensors\_event\_t\* data, int count);//读取输入事件

int batch(int handle, int flags, int64\_t samplingPeriodNs,

int64\_t maxBatchReportLatencyNs);//batch操作，需要模块支持

int flush(int handle); //flush操作，需要模块支持

private:

enum {

batchsensor = 0,

hwmsen,

accel,

magnetic,

gyro,

light,

proximity,

pressure,

temperature,

stepcounter,

pedometer,

in\_pocket,

activity,

pick\_up,

face\_down,

shake,

heartrate,

tilt,

wake\_gesture,

glance\_gesture,

linear\_acceleration,

rotation\_vector,

game\_rotation\_vector,

gravity,

geomagnetic\_rotation\_vector,

bringtosee,

numSensorDrivers,

numFds,

};//mSensors数组索引(index)

//上层请求的sensor ID转换成hal层对应的index，即上面的枚举值

int handleToDriver(int handle) const {

switch (handle) {

case ID\_ACCELEROMETER:

return accel;

case ID\_MAGNETIC:

case ID\_ORIENTATION:

return magnetic;

case ID\_PROXIMITY:

return proximity;

case ID\_LIGHT:

return light;

case ID\_GYROSCOPE:

return gyro;

case ID\_PRESSURE:

return pressure;

case ID\_TEMPRERATURE:

return temperature;

case ID\_STEP\_COUNTER:

case ID\_STEP\_DETECTOR:

case ID\_SIGNIFICANT\_MOTION:

return stepcounter;

case ID\_PEDOMETER:

return pedometer;

case ID\_IN\_POCKET:

return in\_pocket;

case ID\_ACTIVITY:

return activity;

case ID\_PICK\_UP\_GESTURE:

return pick\_up;

case ID\_FACE\_DOWN:

return face\_down;

case ID\_SHAKE:

return shake;

case ID\_HEART\_RATE:

return heartrate;

case ID\_TILT\_DETECTOR:

return tilt;

case ID\_WAKE\_GESTURE:

return wake\_gesture;

case ID\_GLANCE\_GESTURE:

return glance\_gesture;

case ID\_LINEAR\_ACCELERATION:

return linear\_acceleration;

case ID\_ROTATION\_VECTOR:

return rotation\_vector;

case ID\_GAME\_ROTATION\_VECTOR:

return game\_rotation\_vector;

case ID\_GRAVITY:

return gravity;

case ID\_GEOMAGNETIC\_ROTATION\_VECTOR:

return geomagnetic\_rotation\_vector;

case ID\_BRINGTOSEE:

return bringtosee;

default:

break;

//return pressure;

}

return -EINVAL;

}

static const size\_t wake = numFds - 1;//读管道

static const char WAKE\_MESSAGE = 'W';

/\*pollfd是poll()函数操作的对象类型

\*struct pollfd {

\* int fd;

\* short events;

\* short revents;

\*};

\*/

struct pollfd mPollFds[numFds];

int mWritePipeFd;//写管道

SensorBase\* mSensors[numSensorDrivers];//hold所有的sensor

};

构造函数sensors\_poll\_context\_t()实例化各个sensor对象，初始化mPollFds，创建管道

sensors\_poll\_context\_t::sensors\_poll\_context\_t()

{

mSensors[hwmsen] = new Hwmsen();

mPollFds[hwmsen].fd = ((Hwmsen \*)mSensors[hwmsen])->mdata\_fd;

mPollFds[hwmsen].events = POLLIN;

mPollFds[hwmsen].revents = 0;

mSensors[accel] = new AccelerationSensor();

mPollFds[accel].fd = ((AccelerationSensor\*)mSensors[accel])->mdata\_fd;

mPollFds[accel].events = POLLIN;

mPollFds[accel].revents = 0;

//...

int wakeFds[2];

int result = pipe(wakeFds);

fcntl(wakeFds[0], F\_SETFL, O\_NONBLOCK);

fcntl(wakeFds[1], F\_SETFL, O\_NONBLOCK);

mWritePipeFd = wakeFds[1];//write end

mPollFds[wake].fd = wakeFds[0];//read end

mPollFds[wake].events = POLLIN;

mPollFds[wake].revents = 0;

}

activate() enable/disable handle指定的某个sensor并向管道写入唤醒消息(字符’W’)

int sensors\_poll\_context\_t::

activate(int handle, int enabled)

{

//hanlde代表指定enalbe/disable某个sensor

int err=0;

int index = handleToDriver(handle);//获取sensor在mSensors数组内的索引

if(ID\_ORIENTATION == handle){

//方向传感器需要先激活加速度传感器

((AccelerationSensor\*)(mSensors[accel]))->enableNoHALDataAcc(enabled);

((Hwmsen\*)(mSensors[hwmsen]))->enableNoHALDataAcc(enabled);

}

if( (index >= numSensorDrivers) || (index < 0) ){

return 0;

}

if(NULL != mSensors[index])

{

ALOGD( "use new sensor index=%d,

mSensors[index](%x)", index, mSensors[index]);

if(this->device.common.version

>= SENSORS\_DEVICE\_API\_VERSION\_1\_1)

{

ALOGD("support batch active \n" );

err = mSensors[batchsensor]->enable(handle, enabled);

}

err = mSensors[index]->enable(handle, enabled);

}

if(err || index<0 ){

// notify to hwmsen sensor to support old architecture

err = mSensors[hwmsen]->enable(handle, enabled);//enable

}

if (enabled && !err) {

const char wakeMessage(WAKE\_MESSAGE);

int result = write(mWritePipeFd, &wakeMessage, 1);

}

return err;

}

setDelay()设置延迟及轮询周期

int sensors\_poll\_context\_t::

setDelay(int handle, int64\_t ns)

{

int err =0;

int index = handleToDriver(handle);

if( (index >= numSensorDrivers) || (index < 0) )

{

return 0;

}

if(NULL != mSensors[index])

{

err = mSensors[index]->setDelay(handle, ns);

}

if(err || index<0)

{

// notify to hwmsen sensor to support old architecture

err = mSensors[hwmsen]->setDelay(handle, ns);//setDelay

}

return err;

}

pollEvents()调用SensorBase的readEvents()读取sensor数据

int sensors\_poll\_context\_t::

pollEvents(sensors\_event\_t\* data, int count)

{

int nbEvents = 0;

int n = 0;

/\*

\*event事件的读取是调用poll()完成的

\* sensor->readEvents(data, count)会调用processEvent()获取sensor数据

\*/

do {

// see if we have some leftover from the last poll()

for (int i=0 ; count && i<numSensorDrivers ; i++) {

SensorBase\* const sensor(mSensors[i]);

if ((mPollFds[i].revents & POLLIN) || (sensor->hasPendingEvents()))

{

int nb = sensor->readEvents(data, count);//nb : 接收到的事件个数

if (data->type == SENSOR\_TYPE\_PEDOMETER) {

}

if (data->type == SENSOR\_TYPE\_SIGNIFICANT\_MOTION) {

}

if (nb < count) {

// no more data for this sensor

mPollFds[i].revents = 0;

}

for (int j=0;j<nb;j++)

{

if (data[j].type == SENSOR\_TYPE\_META\_DATA)

data[j].meta\_data.sensor += ID\_OFFSET;

else

data[j].sensor += ID\_OFFSET;

}

count -= nb;

nbEvents += nb;

data += nb;

}

}

if (count) {

// we still have some room, so try to see if we can get

// some events immediately or just wait if we don't have

// anything to return

n = poll(mPollFds, numFds, nbEvents ? 0 : -1);//读取事件

if (n<0) {

int err;

err = errno;

ALOGE("poll() failed (%s)", strerror(errno));

return -err;

}

if (mPollFds[wake].revents & POLLIN) {

char msg;

int result = read(mPollFds[wake].fd, &msg, 1);

ALOGE\_IF(result<0, "error reading from wake pipe (%s)",

strerror(errno));

ALOGE\_IF(msg != WAKE\_MESSAGE,

"unknown message on wake queue (0x%02x)", int(msg));

mPollFds[wake].revents = 0;

}

}

// if we have events and space, go read them

} while (n && count);

return nbEvents;

}

batch()与flush()是新版本加入的，具体功能撇开MTK sensor模块后再分析

int sensors\_poll\_context\_t::

batch(int handle, int flags, int64\_t samplingPeriodNs,

int64\_t maxBatchReportLatencyNs)

{

int err=0;

int index = handleToDriver(handle);

if( (index >= numSensorDrivers) || (index < 0) )

{

return 0;

}

if(maxBatchReportLatencyNs == 0)

{

if ((flags & SENSORS\_BATCH\_DRY\_RUN) == 0)//Don't set delay if dry run.

{

err = setDelay(handle, samplingPeriodNs);

}

err = mSensors[index]->batch(handle, flags,

samplingPeriodNs, maxBatchReportLatencyNs);//tell single sensors to disable there own polling and use

}

else

{

err = mSensors[index]->batch(handle, flags,

samplingPeriodNs, maxBatchReportLatencyNs);//tell single sensors to disable there own polling and use

}

if(err)

{

return 0;

}

else

{

// notify to batch sensor to support new architecture

err = mSensors[batchsensor]->batch(handle, flags, samplingPeriodNs, maxBatchReportLatencyNs);

return err;

}

}

int sensors\_poll\_context\_t::

flush(int handle)

{

ALOGD( "flush handle =%d",handle);

int err=0;

int index = handleToDriver(handle);

if( (index >= numSensorDrivers) || (index < 0) )

{

ALOGE("new setDelay handle error(%d)\n",index);

return 0;

}

err = mSensors[index]->flush(handle);

ALOGE("go to batchsensor\n",handle);

// notify to hwmsen sensor to support old architecture

err = mSensors[batchsensor]->flush(handle);

const char wakeMessage(WAKE\_MESSAGE);

int result = write(mWritePipeFd, &wakeMessage, 1);

ALOGE\_IF(result<0, "error sending wake message (%s)", strerror(errno));

return err;

}

* **sensors.c**

硬件抽象层两个重要的数据结构hw\_module\_t及hw\_device\_t需要实现，这样才能通过统一的硬件抽象层接口（libhardware.so）提供的hw\_get\_module\_t()函数返回hw\_module\_t类型对象，继而调用.open返回hw\_device\_t对象。这里的hw\_device\_t对象在nusensors.cpp定义的类sensors\_poll\_context\_t 第一个成员struct sensors\_poll\_device\_1 device内定义，这样通过将返回的hw\_device\_t对象的指针强转成sensors\_poll\_device\_1类型便可操作挂载的函数指针（这些函数指针用来操作sensors\_poll\_context\_t的成员方法）。

下面看下nusensors.cpp关于hw\_device\_t对象的实现：

struct sensors\_poll\_device\_1类型定义如下：

struct sensors\_poll\_device\_1 {

union {

/\* sensors\_poll\_device\_1 is compatible with sensors\_poll\_device\_t,

\* and can be down-cast to it

\*/

struct sensors\_poll\_device\_t v0;//满足兼容性要求

struct {

struct hw\_device\_t common;//硬件抽象层设备类型，必须是第一个成员

int (\*activate)(struct sensors\_poll\_device\_t \*dev,

int sensor\_handle, int enabled);

int (\*setDelay)(struct sensors\_poll\_device\_t \*dev,

int sensor\_handle, int64\_t sampling\_period\_ns);

int (\*poll)(struct sensors\_poll\_device\_t \*dev,

sensors\_event\_t\* data, int count);

};

};

int (\*batch)(struct sensors\_poll\_device\_1\* dev,

int sensor\_handle, int flags, int64\_t sampling\_period\_ns,

int64\_t max\_report\_latency\_ns);

int (\*flush)(struct sensors\_poll\_device\_1\* dev, int sensor\_handle);

void (\*reserved\_procs[8])(void);

} ;

init\_nusensors()被hw\_module\_t对象的hw\_module\_methods\_t挂载的函数open\_sensors()调用，初始化sensors\_poll\_device\_1（hw\_device\_t）类型对象

int

init\_nusensors(hw\_module\_t const\* module, hw\_device\_t\*\* device)

{

int status = -EINVAL;

//实例化sensors\_poll\_context\_t对象，从而创建各个sensor的实例

sensors\_poll\_context\_t \*dev = new sensors\_poll\_context\_t();

memset(&dev->device, 0, sizeof(sensors\_poll\_device\_1));

//hw\_device\_t对象初始化

dev->device.common.tag = HARDWARE\_DEVICE\_TAG;

#if defined(SENSOR\_BATCH\_SUPPORT)

|| defined(CUSTOM\_KERNEL\_SENSORHUB)

dev->device.common.version = SENSORS\_DEVICE\_API\_VERSION\_1\_1;

#else

dev->device.common.version = SENSORS\_DEVICE\_API\_VERSION\_1\_0;

#endif

dev->device.common.module = const\_cast<hw\_module\_t\*>(module);

dev->device.common.close = poll\_\_close;

dev->device.activate = poll\_\_activate;

dev->device.setDelay = poll\_\_setDelay;

dev->device.poll = poll\_\_poll;

dev->device.batch = poll\_\_batch;

dev->device.flush = poll\_\_flush;

\*device = &dev->device.common;

status = 0;

return status;

}

poll\_\_close()等函数定义如下：

static int

poll\_\_close(struct hw\_device\_t \*dev)

{

//将传递的hw\_device\_t对象强转成sensors\_poll\_context\_t类型，继而操作其定义的公有成员，如activate()/setDelay()等

sensors\_poll\_context\_t \*ctx = (sensors\_poll\_context\_t \*)dev;

if (ctx) {

delete ctx;

}

return 0;

}

static int

poll\_\_activate(struct sensors\_poll\_device\_t \*dev, int handle, int enabled)

{

sensors\_poll\_context\_t \*ctx = (sensors\_poll\_context\_t \*)dev;

return ctx->activate(handle-ID\_OFFSET, enabled);

}

static int

poll\_\_setDelay(struct sensors\_poll\_device\_t \*dev, int handle, int64\_t ns)

{

sensors\_poll\_context\_t \*ctx = (sensors\_poll\_context\_t \*)dev;

return ctx->setDelay(handle-ID\_OFFSET, ns);

}

static int

poll\_\_poll(struct sensors\_poll\_device\_t \*dev,sensors\_event\_t\* data, int count)

{

sensors\_poll\_context\_t \*ctx = (sensors\_poll\_context\_t \*)dev;

return ctx->pollEvents(data, count);

}

static int

poll\_\_batch(struct sensors\_poll\_device\_1 \*dev, int handle, int flags,

int64\_t samplingPeriodNs, int64\_t maxBatchReportLatencyNs)

{

sensors\_poll\_context\_t \*ctx = (sensors\_poll\_context\_t \*)dev;

return ctx->batch(handle-ID\_OFFSET, flags, samplingPeriodNs, maxBatchReportLatencyNs);

}

static int

poll\_\_flush(struct sensors\_poll\_device\_1 \*dev, int handle)

{

sensors\_poll\_context\_t \*ctx = (sensors\_poll\_context\_t \*)dev;

return ctx->flush(handle-ID\_OFFSET);

}

sensors.c文件实现了硬件抽象层的接口hw\_module\_t类型的对象

struct sensors\_module\_t类型定义如下：

struct sensors\_module\_t {

struct hw\_module\_t common;//硬件抽象层模块数据结构，必须是第一个成员

/\*\*

\* Enumerate all available sensors. The list is returned in "list".

\* @return number of sensors in the list

\*/

int (\*get\_sensors\_list)(struct sensors\_module\_t\* module,

struct sensor\_t const\*\* list);

};

硬件抽象层模块声明的结构体变量名必须是HAL\_MODULE\_INFO\_SYM

struct sensors\_module\_t HAL\_MODULE\_INFO\_SYM = {

//初始化hw\_module\_t

.common = {

.tag = HARDWARE\_MODULE\_TAG,//tag必须是这个名字

.version\_major = 1,

.version\_minor = 0,

.id = SENSORS\_HARDWARE\_MODULE\_ID,//模块id

.name = "MTK SENSORS Module",

.author = "Mediatek",

.methods = &sensors\_module\_methods,// hw\_module\_methods\_t类型

.get\_sensors\_list = sensors\_\_get\_sensors\_list,//获得设备所有的支持的sensor，代码很简单返回指向数组的指针

};

硬件抽象层模块的操作方法在hw\_module\_methods\_t类型对象内：

static struct hw\_module\_methods\_t sensors\_module\_methods = {

.open = open\_sensors//调用nusensors.cpp定义的init\_sensors()函数

};

static int open\_sensors(const struct hw\_module\_t\* module, const char\* name,

struct hw\_device\_t\*\* device)

{

return init\_nusensors(module, device);

}

hal层模块编译成动态库的形式（路径/system/lib/hw/sensors.<platform>.so），有通用的hal层动态库/system/lib/libhardware.so通过传递模块ID给（如sensor的SENSORS\_HARDWARE\_MODULE\_ID）hw\_get\_module\_t()获取返回的hw\_module\_t与hw\_device\_t对象继而获取sensor的操作。