



Unisoc Confidential For hiar

SensorHub 动态加载驱动指导

文档版本
发布日期

V1.3
2020-10-22

版权所有 © 紫光展锐（上海）科技有限公司。保留一切权利。

本文件所含数据和信息都属于紫光展锐（上海）科技有限公司（以下简称紫光展锐）所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负任何与本文件相关的直接或间接的、任何伤害或损失。

请参照交付物中说明文档对紫光展锐交付物进行使用，任何人对紫光展锐交付物的修改、定制化或违反说明文档的指引对紫光展锐交付物进行使用造成的任何损失由其自行承担。紫光展锐交付物中的性能指标、测试结果和参数等，均为在紫光展锐内部研发和测试系统中获得的，仅供参考，若任何人需要对交付物进行商用或量产，需要结合自身的软硬件测试环境进行全面的测试和调试。

Unisoc Confidential For hiar

紫光展锐（上海）科技有限公司



前言

概述

该文档适用于展锐当前带 SensorHub 平台的产品，主要介绍 SensorHub 模块动态加载机制、动态加载机制 RTOS 接口、动态加载机制 API 接口、动态加载驱动机制编译及伪代码实现。

读者对象


本文档主要供 Sensor 模块相关技术人员及展锐客户参考使用。

缩略语

缩略语	英文全名	中文解释
OPCODE	Open Code	在 Snsors HAL 层对 Sensor 进行参数配置的文件。
RTOS	Real Time Operating System	实时操作系统：是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理的操作系统。
DS-5	Arm Development Studio 5	ARM 公司最新推出的，支持包括 ARMv8 架构在内所有 ARM 内核的嵌入式软件开发工具。

符号约定

在本文中可能出现下列标志，它所代表的含义如下。

符号	说明
 说明	用于突出重要/关键信息、补充信息和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

变更信息

文档版本	发布日期	修改说明
V1.0	2019-10-25	初稿。
V1.1	2019-11-15	对初审问题进行修正。
V1.2	2020-01-19	适用产品信息增加 UIS8581E 平台。
V1.3	2020-10-22	<ul style="list-style-type: none">技术更新：删除原文档中动态加载驱动机制编译相关内容，并以文档《EmBitz 编译 Sensorhub 动态加载驱动介绍》代替。格式调整，语言描述优化。

关键字

SensorHub、动态加载驱动。

Unisoc Confidential For hiar

目 录

1 背景介绍.....	1
2 动态加载机制 RTOS 接口.....	2
2.1 init param 接口.....	2
2.2 i2c read 接口.....	3
2.3 i2c write 接口.....	4
2.4 debug_trace 接口.....	5
2.5 msleep 接口.....	5
2.6 i2c open 接口.....	5
2.7 i2c close 接口.....	6
2.8 gpio control 接口.....	7
3 动态加载机制 API 接口.....	8
3.1 磁力计 API.....	9
3.1.1 mag_init 接口.....	9
3.1.2 mag_update 接口.....	9
3.1.3 mag_close 接口.....	10
3.1.4 mag_update_cfg 接口.....	11
3.2 加速度计 API.....	11
3.2.1 acc_enable 接口.....	11
3.2.2 acc_get_data 接口.....	12
3.2.3 acc_disable 接口.....	13
3.2.4 acc_update_cfg 接口.....	13
3.3 陀螺仪 API.....	14
3.3.1 gyro_enable 接口.....	14
3.3.2 gyro_get_data 接口.....	14
3.3.3 gyro_disable 接口.....	15
3.3.4 gyro_update_cfg 接口.....	15
3.4 压力计 API.....	16
3.4.1 pressure_enable 接口.....	16
3.4.2 pressure_get_data 接口.....	17
3.4.3 pressure_disable 接口.....	17
3.4.4 pressure_update_cfg 接口.....	18
3.5 光线传感器 API.....	18
3.5.1 light_enable 接口.....	18
3.5.2 light_get_data 接口.....	19
3.5.3 light_disable 接口.....	20

3.5.4 light_update_cfg 接口	20
3.6 距离传感器 API.....	21
3.6.1 proximity_enable 接口	21
3.6.2 proximity_get_data 接口	22
3.6.3 proximity_disable 接口	22
3.6.4 proximity_update_cfg 接口.....	23
4 动态加载驱动机制编译及伪代码实现.....	24
4.1 动态加载驱动机制编译	24
4.2 动态加载驱动机制伪代码实现	24
4.2.1 动态加载机制使用说明	24
4.2.2 动态加载文件伪代码实现	24

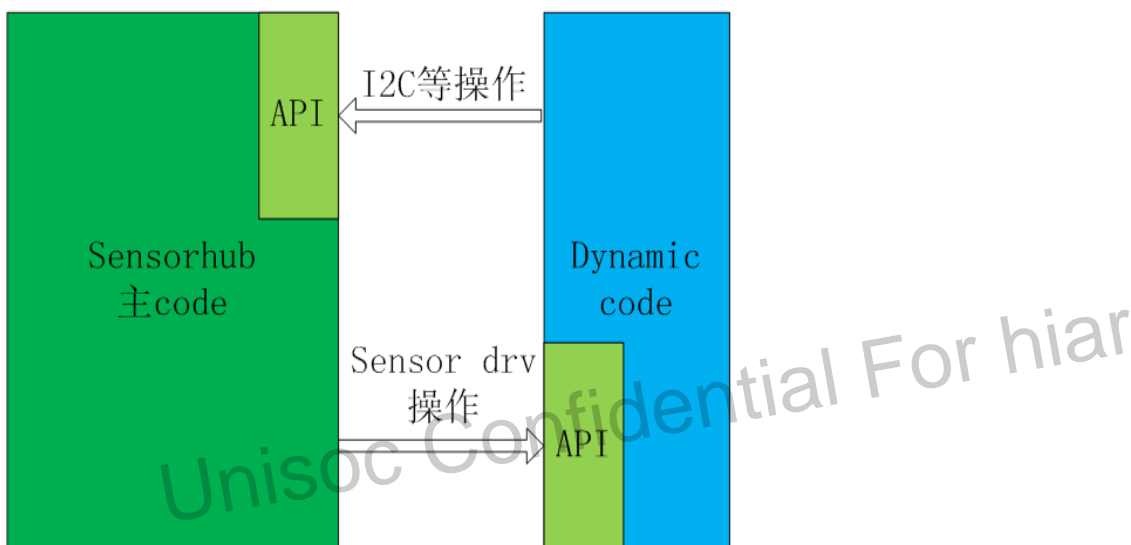
Unisoc Confidential For hiar

1 背景介绍

在实际开发中，有些 Sensor 需要搭配特定的算法才能正常工作。例如磁力计，其 raw data 需要进行校准才能正常使用，而不同厂家的校准算法各有不同，很难达到统一。又例如距离传感器的快速校准算法，每个厂家都有自己的一套逻辑。针对这种情况，展锐推出动态加载机制，解决各厂家之间的差异问题。

动态加载机制的调用示意图如图 1-1 所示。

图1-1 动态加载机制示意图



动态加载机制和 SensorHub 的主 code 分开编译，两者通过定义好的 API 来相互调用，所以只要 dynamic code 中 API 的地址和定义不变，无论其里面内容是什么，对 SensorHub 的主 code 来讲都是透明的。

说明

使用动态加载驱动机制前，请 Sensor 驱动开发人员参阅适用自己开发平台和软件版本的 SensorHub 客制化配置（SensorHub 客制化指导）文档，了解 Sensors HAL 对各类型 Sensor 参数的配置以及使用动态加载机制时如何配置相关参数。

2 动态加载机制 RTOS 接口

动态加载机制与 SensorHub 的主 code 分开编译，在动态加载机制中无法直接调用 RTOS 中的相应接口，所以只能通过回调函数的方式来使用。目前为动态加载机制开放的系统接口如下所示：

```
struct patch_table_dynamic
{
uint32 (*init_param)(int32 *param, int size);
uint32 (*debug_trace)(const char *x_format, ...);
uint32 (*extern_i2c_read)(uint32 handle, uint8 *reg_addr, uint8 *buffer, uint32 bytes);
uint32 (*extern_i2c_write)(uint32 handle, uint8 *reg_addr, uint8 *buffer, uint32 bytes);
void (*msleep)(unsigned long ms);
uint32 (*extern_i2c_open)(I2C_DEV *dev);
uint32 (*extern_i2c_close)(uint32 handle);
void (*extern_gpio_ctrl)( uint32 num, BOOLEAN value);
};
```

下面对这些接口使用进行介绍。

2.1 init param 接口

【函数功能】

获取所支持的传感器（Sensor）的 i2c handle。i2c handle 用于读写 Sensor 寄存器时使用，目前一共支持 6 种实体 Sensor，支持的 Sensor 类型定义如下：

```
enum sensor_type
{
    ACC_TYPE,
    MAG_TYPE,
    GYRO_TYPE,
    PRO_TYPE,
    LIGHT_TYPE,
    PRESSURE_TYPE,
};
```

【函数原型】

其函数指针声明如下所示：

```
uint32 (*init_param)(int32 *param, int size);
```


【参数说明】

参数	说明
param	指向存放 Sensor i2c handle 数据地址的指针。
size	param 的大小。

【返回值】

- 成功：返回 0。
- 失败：返回-1。

【示例】

获取所支持的传感器的 i2c handle。

```
int32 i2c_handle[6];  
shub_extern_call->init_param(i2c_handle, sizeof(i2c_handle));
```

2.2 i2c read 接口

【函数功能】

I2C 读函数。

【函数原型】

其函数指针声明如下所示：

```
uint32 (*extern_i2c_read)(uint32 handle, uint8 *reg_addr, uint8 *buffer, uint32 bytes);
```

【参数说明】

参数	说明
handle	调用 init_param 或 extern_i2c_open 获取到的 I2C 从设备。
reg_addr	I2C 从设备的内部寄存器地址。
buffer	存储数据的缓存，从 I2C 从设备读取。
bytes	读取数据所占用的字节数量。

【返回值】

- 成功：返回读取数据占用的字节数量。

- 失败：返回 0。

【示例】

读取 mag sensor 寄存器 0x0F 的数据。

```
uint8 buf[2] = {0x0F,0x00};
ret=shub_extern_call->extern_i2c_read(i2c_handle[MAG_TYPE],&buf[0],&buf[1],1);
```

📖 说明

读取 mag sensor 寄存器值，此处直接使用 mag sensor 的 handle：i2c_handle[MAG_TYPE]即可。

2.3 i2c write 接口

【函数功能】

I2C 写函数。

【函数原型】

其函数指针声明如下所示：

```
uint32 (*extern_i2c_write)(uint32 handle, uint8 *reg_addr, uint8 *buffer, uint32 bytes);
```

【参数说明】

参数	说明
handle	调用 init_param 或 extern_i2c_open 读取到的 I2C 从设备。
reg_addr	I2C 从设备的内部寄存器地址。
buffer	存储数据的缓存，写入 I2C 从设备。
bytes	写入数据所占用的字节数量。

【返回值】

- 成功：返回写入数据占用的字节数量。
- 失败：返回 0。

【示例】

向 mag sensor 寄存器 0x0E 写入 0x38。

```
uint8 buf[2] = {0x0E,0x38};
ret=shub_extern_call->extern_i2c_write(i2c_handle[MAG_TYPE],&buf[0],&buf[1],1);
```

2.4 debug_trace 接口

【函数功能】

打印 log。

【函数原型】

其函数指针声明如下所示：

```
uint32 (*debug_trace)(const char *x_format, ...);
```

2.5 msleep 接口

【函数功能】

休眠接口。

【函数原型】

其函数指针声明如下所示：

```
void (*msleep)(unsigned long ms);
```

【参数说明】

参数	说明
ms	精度，单位毫秒。

【返回值】

无。

【示例】

休眠 10ms。

```
shub_extern_call->msleep(10);
```

2.6 i2c open 接口

【函数功能】

I2C 打开函数。

【函数原型】

其函数指针声明如下所示：

```
uint32 (*extern_i2c_open)(I2C_DEV *dev);
```

【参数说明】

参数	说明
dev	I2C 从设备指针。

【返回值】

- 成功：返回一个 handle，其值大于等于 0。
- 失败：返回-1。

【示例】

打开某一类型 sensor，获取该类型 sensor 的 handle。

```
I2C_DEV dev;  
ret=shub_extern_call->extern_i2c_open(&dev);
```

2.7 i2c close 接口

【函数功能】

I2C 关闭函数。

【函数原型】

其函数指针声明如下所示：

```
uint32 (*extern_i2c_close)(uint32 handle);
```

【参数说明】

参数	说明
handle	调用 init_param 或 extern_i2c_open 获取到的 I2C 从设备。

【返回值】

- 成功：返回 0。
- 失败：返回-1。

【示例】

关闭 mag sensor。

```
ret=shub_extern_call->extern_i2c_close(i2c_handle[MAG_TYPE]);
```

2.8 gpio control 接口

【函数功能】

GPIO 控制函数，设置相应 GPIO 的高低电平。

【函数原型】

其函数指针声明如下所示：

```
void (*extern_gpio_ctrl)( uint32 num, BOOLEAN value);
```

【参数说明】

参数	说明
num	GPIO ID。
value	0 或 1。

【返回值】

无。

【示例】

设置 gpio58 为高电平。

```
shub_extern_call->extern_gpio_ctrl(58, 1);
```

3 动态加载机制 API 接口

整个动态加载机制可使用的空间只有 20K，所以能用 opcode 完成的部分尽量不用动态加载机制。目前 SensorHub 一共支持六种实体 Sensor，每颗 Sensor 都有四个独立的 API 供开发者使用，这些 API 的定义如下所示：

```
struct patch_table_sensor
{
    int (*mag_init)(int32 sampleTime_ms, float *mag_offset);
    int (*mag_update)(double *mag_raw_data, double *acc, double *gyro, float *cali_mag, int *mag_accuracy, unsigned long currTime, uint32 odr);
    int (*mag_close)(float *mag_offset);
    int (*mag_update_cfg)(int *arg);
    int (*prox_enable)(int mode, void *pdat, int len);
    int (*prox_update_cfg)(int *data);
    int (*prox_get_data)(int *value, float *raw_data);
    int (*prox_disable)(void);
    int (*light_enable)(int mode, void *pdat, int len);
    int (*light_update_cfg)(int *data);
    int (*light_get_data)(int *value, float *raw_data);
    int (*light_disable)(void);
    int (*pressure_enable)(int mode, void *pdat, int len);
    int (*pressure_update_cfg)(int *data);
    int (*pressure_get_data)(int *value, float *raw_data);
    int (*pressure_disable)(void);
    int (*acc_enable)(int mode, void *pdat, int len);
    int (*acc_update_cfg)(int *data);
    int (*acc_get_data)(int *value, float *raw_data);
    int (*acc_disable)(void);
    int (*gyro_enable)(int mode, void *pdat, int len);
    int (*gyro_update_cfg)(int *data);
    int (*gyro_get_data)(int *value, float *raw_data);
    int (*gyro_disable)(void);
    int (*extern_main)(void);
};
```

extern_main 接口是在识别到有加载动态驱动时执行的第一个功能接口，用于将动态加载驱动 ZI 段(bss 段)清零，以及拷贝 sensor i2c handle 的操作。如有需要可以增加一些自定义的操作，否则无需修改。API 接口根据实际情况来定义，如果不需要直接写 NULL 即可。

3.1 磁力计 API

磁力计（Mag sensor）一共有四个 API，其分别会在 mag sensor driver 的相应接口中被调用，这四个 API 的定义分别如下所示：

```
int (*mag_init)(int32 sampleTime_ms, float *mag_offset);
int (*mag_update)(double *mag_raw_data, double *acc, double *gyro, float *cali_mag, int *mag_accuracy, unsigned long long currTime, uint32 odr);
int (*mag_close)(float *mag_offset);
int (*mag_update_cfg)(int *arg);
```

3.1.1 mag_init 接口

【函数功能】

使能 mag sensor 并初始化厂商算法，在每次使能 mag sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*mag_init)(int32 sampleTime_ms, float *mag_offset);
```

【参数说明】

参数	说明
sampleTime_ms	Mag sensor 的采样周期，目前有四档：10ms、20ms、62ms、200ms。
mag_offset	此参数为上一次关闭 mag sensor 且校准的精度值为 3 时所保存下来的 offset 参数，其 size 最大为 24 bytes。如果其值都为 0，则表示之前 mag sensor 的校准的精度值没有达到过 3，或是还没有使用过 mag sensor。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.1.2 mag_update 接口

【函数功能】

传入 acc、Gyro、Mag sensor 的 raw data，获取磁力计校准之后的数据。每次读取 raw data 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*mag_update)(double *mag_raw_data, double *acc, double *gyro, float *cali_mag, int *mag_accuracy, unsigned long long currTime, uint32 odr);
```

【参数说明】

参数	说明
mag_raw_data	mag_raw_data 是有 3 个成员的数组指针，mag_raw_data[0]、mag_raw_data[1]、mag_raw_data[2]分别存放的是未校准的 mag sensor 的 X/Y/Z 轴数据。此数据的单位为 uT。 如果调试的 sensor 比较特殊，可以使用开放出来的 I2C 的接口，自己读取 raw data 进行相应的转换与计算。在转换完之后，需要把 uT 为单位的数据赋值给此参数，后续 Magnetic Uncalibrated 类型的 sensor 数据就是从此参数获取。
acc	acc 是有 3 个成员的数组指针，存放加速度的 raw data，单位为 m/s ² 。
gyro	gyro 是有 3 个成员的数组指针，存放陀螺仪的 raw data，单位为 rad/s。
cali_mag	cali_mag 是有 3 个成员的数组指针，磁力计校准之后的 X/Y/Z 轴数据分别放到 cali_mag[0]、cali_mag[1]、cali_mag[2]，此数据以 uT 为单位。
mag_accuracy	磁力计校准后的精度值。
currTime	当前系统的时间，单位为 ms。
odr	当前 mag sensor 的 odr。

【返回值】

- 正常：返回 0。
- 失败：返回 -1。

3.1.3 mag_close 接口

【函数功能】

关闭 mag sensor，每次关闭 mag sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*mag_close)(float *mag_offset);
```

【参数说明】

参数	说明
mag_offset	此参数为获取 mag sensor 当前的 offset 等参数，和 init 的时候的操作相对应，最大 size 为 24 bytes。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.1.4 mag_update_cfg 接口

【函数功能】

传递软磁校准参数，其 size 为 36 个 int 数据，在 AP 侧配置，调试时倘若需要传递其他参数，也可以复用此接口。每次使能 mag sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*mag_update_cfg)(int *arg);
```

【参数说明】

参数	说明
arg	mag sensor 软磁校准参数，其 size 为 36 个 int 数据，在 AP 侧配置。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.2 加速度计 API

加速度计（Acc sensor）一共有四个 API，其分别会在 acc sensor driver 的相应接口中被调用，这四个 API 的定义分别如下所示：

```
int (*acc_enable)(int mode, void *pdata, int len);  
int (*acc_update_cfg)(int *data);  
int (*acc_get_data)(int *value, float *raw_data);  
int (*acc_disable)(void);
```

3.2.1 acc_enable 接口

【函数功能】

使能 acc sensor，每次使能 acc sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*acc_enable)(int mode, void *pdat, int len);
```

【参数说明】

参数	说明
mode	在 opcode 中配置的 position。
pdat	工厂校准中获得的三轴 offset 参数，其已经转换成 android 坐标系，单位是 m/s ² 。
len	pdat 的长度。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.2.2 acc_get_data 接口

【函数功能】

获取 acc sensor raw data，每次读取 raw data 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*acc_get_data)(int *value, float *raw_data);
```

【参数说明】

参数	说明
value	目前传入的为 NULL。
raw_data	raw_data 是有 3 个成员的数组指针，倘若需要动态加载机制来获取 acc raw data，那把计算好的 X/Y/Z 轴数据分别放到 raw_data[0]、raw_data[1]、raw_data[2]，此数据以 m/s ² 为单位。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.2.3 acc_disable 接口

【函数功能】

关闭 acc sensor，每次关闭 acc sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*acc_disable)(void);
```

【参数说明】

无。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.2.4 acc_update_cfg 接口

【函数功能】

更新当前 acc sensor 的采样周期。

【函数原型】

其函数指针声明如下所示：

```
int (*acc_update_cfg)(int *data);
```

【参数说明】

参数	说明
data	Acc sensor 的采样周期，目前有四档：10ms、20ms、60ms、200ms。 <ul style="list-style-type: none">• data = 0: 设置该 sensor 的采样周期为 10ms。• data = 1: 设置该 sensor 的采样周期为 20ms。• data = 2: 设置该 sensor 的采样周期为 60ms。• data = 3: 设置该 sensor 的采样周期为 200ms。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.3 陀螺仪 API

陀螺仪（Gyro sensor）一共有四个 API，其分别会在 gyro sensor driver 的相应接口中被调用，这四个 API 的定义分别如下所示：

```
int (*gyro_enable)(int mode, void *pdat, int len);
int (*gyro_update_cfg)(int *data);
int (*gyro_get_data)(int *value, float *raw_data);
int (*gyro_disable)(void);
```

3.3.1 gyro_enable 接口

【函数功能】

使能 gyro sensor，每次使能 gyro sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*gyro_enable)(int mode, void *pdat, int len);
```

【参数说明】

参数	说明
mode	在 opcode 中配置的 position。
pdat	工厂校准中获得的三轴 offset 参数，其已经转换成 Android 坐标系，单位是 rad/s。
len	pdat 的长度。

【返回值】

- 正常：返回 0。
- 失败：返回 -1。

3.3.2 gyro_get_data 接口

【函数功能】

获取 gyro sensor raw data，每次读取 raw data 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*gyro_get_data)(int *value, float *raw_data);
```

【参数说明】

参数	说明
value	目前传入的为 NULL。
raw_data	raw_data 是有 3 个成员的数组指针，倘若需要动态加载机制来获取 gyro raw data，那把计算好的 X/Y/Z 轴数据分别放到 raw_data[0]、raw_data[1]、raw_data[2]，此数据以 rad/s 为单位。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.3.3 gyro_disable 接口

【函数功能】

关闭 gyro sensor，每次关闭 gyro sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*gyro_disable)(void);
```

【参数说明】

无。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.3.4 gyro_update_cfg 接口

【函数功能】

更新当前 gyro sensor 的采样周期。

【函数原型】

其函数指针声明如下所示：

```
int (*gyro_update_cfg)(int *data);
```

【参数说明】

参数	说明
data	gyro sensor 的采样周期，目前有四档：10ms、20ms、60ms、200ms。 <ul style="list-style-type: none"> data = 0: 设置该 sensor 的采样周期为 10ms。 data = 1: 设置该 sensor 的采样周期为 20ms。 data = 2: 设置该 sensor 的采样周期为 60ms。 data = 3: 设置该 sensor 的采样周期为 200ms。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.4 压力计 API

压力计（Pressure sensor）一共有四个 API，其分别会在 pressure sensor driver 的相应接口中被调用，这四个 API 的定义分别如下所示：

```
int (*pressure_enable)(int mode, void *pdat, int len);
int (*pressure_update_cfg)(int *data);
int (*pressure_get_data)(int *value, float *raw_data);
int (*pressure_disable)(void);
```

3.4.1 pressure_enable 接口

【函数功能】

使能 pressure sensor，每次使能 pressure sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*pressure_enable)(int mode, void *pdat, int len);
```

【参数说明】

参数	说明
mode	在 opcode 中配置的 position。
pdat	工厂校准中获得的三轴 offset 参数，其已经转换成 Android 坐标系，单位是 hpa。
len	pdat 的长度。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.4.2 pressure_get_data 接口

【函数功能】

获取 pressure sensor raw data，每次读取 raw data 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*pressure_get_data)(int *value, float *raw_data);
```

【参数说明】

参数	说明
value	目前传入的为 NULL。
raw_data	raw_data 是有 3 个成员的数组指针，倘若需要动态加载机制来获取 pressure raw data，那把计算好的数据放到 raw_data[0]，此数据以 hpa 为单位。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.4.3 pressure_disable 接口

【函数功能】

关闭 pressure sensor，每次关闭 pressure sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*pressure_disable)(void);
```

【参数说明】

无。

【返回值】

- 正常：返回 0。

- 失败：返回-1。

3.4.4 pressure_update_cfg 接口

【函数功能】

更新当前 pressure sensor 的采样周期。

【函数原型】

其函数指针声明如下所示：

```
int (*pressure_update_cfg)(int *data);
```

【参数说明】

参数	说明
data	pressure sensor 的采样周期，目前有四档：10ms、20ms、60ms、200ms。 <ul style="list-style-type: none"> • data = 0: 设置该 sensor 的采样周期为 10ms。 • data = 1: 设置该 sensor 的采样周期为 20ms。 • data = 2: 设置该 sensor 的采样周期为 60ms。 • data = 3: 设置该 sensor 的采样周期为 200ms。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.5 光线传感器 API

光线传感器（Light sensor）一共有四个 API，其分别会在 light sensor driver 的相应接口中被调用，这四个 API 的定义分别如下所示：

```
int (*light_enable)(int mode, void *pdat, int len);
int (*light_update_cfg)(int *data);
int (*light_get_data)(int *value, float *raw_data);
int (*light_disable)(void);
```

3.5.1 light_enable 接口

【函数功能】

使能 light sensor，每次使能 light sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*light_enable)(int mode, void *pdat, int len);
```

【参数说明】

参数	说明
mode	目前传入的为 NULL。
pdat	目前传入的为 NULL。
len	目前传入的为 NULL。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.5.2 light_get_data 接口

【函数功能】

获取 light sensor raw data，每次读取 raw data 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*light_get_data)(int *value, float *raw_data);
```

【参数说明】

参数	说明
value	目前传入的为 NULL。
raw_data	raw_data 是有 3 个成员的数组指针，倘若需要动态加载机制来获取光照强度，那把计算好的数据放到 raw_data[0]，此数据以 als 为单位。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.5.3 light_disable 接口

【函数功能】

关闭 light sensor，每次关闭 light sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*light_disable)(void);
```

【参数说明】

无。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.5.4 light_update_cfg 接口

【函数原型】

其函数指针声明如下所示：

```
int (*light_update_cfg)(int *data);
```

【函数功能】

更新当前 light sensor 的采样周期。

【参数说明】

参数	说明
data	light sensor 的采样周期，目前有四档：10ms、20ms、60ms、200ms。 <ul style="list-style-type: none">• data = 0：设置该 sensor 的采样周期为 10ms。• data = 1：设置该 sensor 的采样周期为 20ms。• data = 2：设置该 sensor 的采样周期为 60ms。• data = 3：设置该 sensor 的采样周期为 200ms。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.6 距离传感器 API

距离传感器（Proximity sensor）一共有四个 API，其分别会在 proximity sensor driver 的相应接口中被调用，这四个 API 的定义分别如下所示：

```
int (*prox_enable)(int mode, void *pdat, int len);
int (*prox_update_cfg)(int *data);
int (*prox_get_data)(int *value, float *raw_data);
int (*prox_disable)(void);
```

3.6.1 proximity_enable 接口

【函数功能】

使能 prox sensor，每次使能 prox sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*prox_enable)(int mode, void *pdat, int len);
```

【参数说明】

参数	说明
mode	工作模式： <ul style="list-style-type: none"> 1：工厂校准模式。 0：正常工作模式。
pdat	传入的是工厂校准获得的参数。 <ul style="list-style-type: none"> Byte3-Byte0：自动校准的 calibrator(自动校准获取的底噪值)。 Byte7-Byte4：手动校准(3cm)的 calibrator(手动校准获取的接近门限值)。 Byte11-Byte8：手动校准(5cm)的 calibrator(手动校准获取的远离门限值)。 Byte12：1 为自动校准(只校准底噪)，6 为手动校准(校准接近门限和远离门限)。
len	pdat 的长度。

【返回值】

- 正常：返回 0。
- 失败：返回-1。

3.6.2 proximity_get_data 接口

【函数功能】

获取 prox sensor raw data，每次读取 raw data 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*prox_get_data)(int *value, float *raw_data);
```

【参数说明】

参数	说明
value	目前传入的为 NULL。
raw_data	raw_data 是有 3 个成员的数组指针，需要将计算出的距离值或者接近(0)/远离(5)状态值放到 raw_data[0]，得到的距感采样值放到 raw_data[2]。

【返回值】

- 正常：返回 0。
- 失败：返回 -1。

3.6.3 proximity_disable 接口

【函数功能】

关闭 prox sensor，每次关闭 prox sensor 时都会调用该接口。

【函数原型】

其函数指针声明如下所示：

```
int (*prox_disable)(void);
```

【参数说明】

无。

【返回值】

- 正常：返回 0。
- 失败：返回 -1。

3.6.4 proximity_update_cfg 接口

【函数功能】

更新当前 prox sensor 的采样周期。

【函数原型】

其函数指针声明如下所示：

```
int (*prox_update_cfg)(int *data);
```

【参数说明】

参数	说明
Data	prox sensor 的采样周期，目前有四档：10ms、20ms、60ms、200ms。 data = 0: 设置该 sensor 的采样周期为 10ms。 data = 1: 设置该 sensor 的采样周期为 20ms。 data = 2: 设置该 sensor 的采样周期为 60ms。 data = 3: 设置该 sensor 的采样周期为 200ms。

【返回值】

- 正常：返回 0。
- 失败：返回 -1。

4 动态加载驱动机制编译及伪代码实现

4.1 动态加载驱动机制编译

动态加载驱动编译方法请参考《EmBitz 编译 Sensorhub 动态加载驱动介绍》。

4.2 动态加载驱动机制伪代码实现

动态加载驱动机制可以配合 OPCODE 方式实现某一类型 Sensor 的所有功能，亦可以独立实现某一类型 Sensor 的所有功能。有关 OPCODE 方式的详细信息，请参阅适用自己开发平台和软件版本的 SensorHub 客制化配置（SensorHub 客制化指导）文档。Sensor 正常工作需要并且只执行一次的操作就可以使用默认的 OPCODE 方式实现。

文档最后一节，展锐将以伪代码的形式实现动态加载文件 patch_table_sensor.c 供 Sensor 模块相关技术人员参考。一般来讲各个厂家的地磁 Sensor 都具有自己的校准算法，因此本节会以地磁 Sensor 为例实现动态加载驱动机制的伪代码。

4.2.1 动态加载机制使用说明

SensorHub 驱动的实现流程是先执行 OPCODE 方式，若满足动态加载驱动机制条件那么会再调用动态加载驱动机制中的相关接口。

以距感 Sensor 为例，有些客户会使用自研算法获取接近远离门限值并且得到某一时刻的接近远离状态值。此时客户便可根据本文档的说明只实现 prox_enable()和 prox_get_data()接口，其他 API 接口直接写 NULL 即可。赋值为 NULL 的 API 接口会使用默认的(OPCODE)方式。

4.2.2 动态加载文件伪代码实现

4.2.2.1 动态加载驱动头文件 overlay_types.h

```
#ifndef _OVERLAY_TYPES_H_
#define _OVERLAY_TYPES_H_

typedef unsigned int uint32;
typedef signed int int32;

typedef unsigned long long uint64;
typedef long long int64;

typedef unsigned char uint8;
typedef signed char int8;
```

```
typedef unsigned short uint16;
typedef signed short int16;

extern const volatile struct patch_table_dynamic * const shub_extern_call;
extern int32 i2c_handle[6];

#endif
```

4.2.2.2 动态加载驱动文件 patch_table_sensor.c

```
#include <stdio.h>
#include <string.h>
#include "overlay_types.h"

enum sensor_type
{
    ACC_TYPE,
    MAG_TYPE,
    GYRO_TYPE,
    PRO_TYPE,
    LIGHT_TYPE,
    PRESSURE_TYPE,
};

struct patch_table_sensor
{
    int (*mag_init)(int32 sampleTime_ms, float *mag_offset);
    int (*mag_update)(double *mag_raw_data, double *acc, double *gyro, float *cali_mag, int *mag_accuracy, unsigned long long currTime, uint32 odr);
    int (*mag_close)(float *mag_offset);
    int (*mag_update_cfg)(int *arg);
    int (*prox_enable)(int mode, void *pdat, int len);
    int (*prox_update_cfg)(int *data);
    int (*prox_get_data)(int *value, float *raw_data);
    int (*prox_disable)(void);
    int (*light_enable)(int mode, void *pdat, int len);
    int (*light_update_cfg)(int *data);
    int (*light_get_data)(int *value, float *raw_data);
    int (*light_disable)(void);
    int (*pressure_enable)(int mode, void *pdat, int len);
    int (*pressure_update_cfg)(int *data);
};
```

```
int (*pressure_get_data)(int *value, float *raw_data);
int (*pressure_disable)(void);
int (*acc_enable)(int mode, void *pdat, int len);
int (*acc_update_cfg)(int *data);
int (*acc_get_data)(int *value, float *raw_data);
int (*acc_disable)(void);
int (*gyro_enable)(int mode, void *pdat, int len);
int (*gyro_update_cfg)(int *data);
int (*gyro_get_data)(int *value, float *raw_data);
int (*gyro_disable)(void);
int (*extern_main)(void);
};

struct patch_table_dynamic
{
uint32 (*init_param)(int32 *param, int size);
uint32 (*debug_trace)(const char *x_format, ...);
uint32 (*extern_i2c_read)(uint32 handle, uint8 *reg_addr, uint8 *buffer, uint32 bytes);
uint32 (*extern_i2c_write)(uint32 handle, uint8 *reg_addr, uint8 *buffer, uint32 bytes);
void (*msleep)(unsigned long ms);
uint32 (*extern_i2c_open)(I2C_DEV *dev);
uint32 (*extern_i2c_close)(uint32 handle);
void (*extern_gpio_ctrl)(uint32 num, BOOLEAN value);
};

extern uint32 Image$$EXEC_CALIBRATE_SENSOR_IMAGE$$ZI$$Base;
extern uint32 Image$$EXEC_CALIBRATE_SENSOR_IMAGE$$ZI$$Limit;

static uint32 sensor_library_zi_Limit = &Image$$EXEC_CALIBRATE_SENSOR_IMAGE$$ZI$$Limit;
static uint32 sensor_library_zi_Base = &Image$$EXEC_CALIBRATE_SENSOR_IMAGE$$ZI$$Base;

const volatile struct patch_table_dynamic * const shub_extern_call =
    (volatile struct patch_table_dynamic *) (0x3AFA0);
/* ACC sensor的handle: i2c_handle[ACC_TYPE]或i2c_handle[0]
MAG sensor的handle: i2c_handle[MAG_TYPE]或i2c_handle[1]
GYRO sensor的handle: i2c_handle[GYRO_TYPE]或i2c_handle[2]
PRO sensor的handle: i2c_handle[PRO_TYPE]或i2c_handle[3]
LIGHT sensor的handle: i2c_handle[LIGHT_TYPE]或i2c_handle[4]
PRESSURE sensor的handle: i2c_handle[PRESSURE_TYPE]或i2c_handle[5]
*/
int32 i2c_handle[6];
```



```
/* InitMain()接口是在识别到有加载动态驱动时执行的第一个功能接口，用于将动态加载驱动ZI段(bss段)清零，以及拷贝
sensor i2c handle的操作；如有需要可以增加一些自定义的操作，否则无需修改
*/
static int InitMain(void)
{
    memset(sensor_library_zi_Base, 0, sensor_library_zi_Limit - sensor_library_zi_Base);
    shub_extern_call->init_param(i2c_handle, sizeof(i2c_handle));

    return 0;
}

/* PatchTableMagInit()根据需要进行实现，函数参数根据实际需要使用时 */
static int PatchTableMagInit(int32 sampleTime_ms, float *mag_offset)
{
    /* enable时调用，地磁sensor正常工作所需要（例如：初始化sensor厂商的地磁算法，可以不使用函数参数）的操作在此函数中实现。
    */
    //向mag sensor地址为0X00的寄存器中写入0X00完成地磁算法的初始化
    uint8 buf[2] = {0x00, 0x00};
    uint32 ret = 0;

    shub_extern_call->debug_trace("Mag sensor init enter\n");
    ret = shub_extern_call->extern_i2c_write(i2c_handle[MAG_TYPE], &buf[0], &buf[1], 1);
    if(ret == 0){
        shub_extern_call->debug_trace("Mag sensor init fail!\n");
        return -1;
    }

    return 0;
}

/* PatchTableMagUpdate()根据需要进行实现，函数参数根据实际需要使用时 */
static int PatchTableMagUpdate(double *mag_raw_data, double *acc, double *gyro,
                                float *cali_mag, int *mag_accuracy, uint64 currTime, uint32 odr)
{
    /* get_raw_data时调用，地磁sensor正常工作时得到校准数据及校准精度在此函数中实现 */
    /* 例如：获取地磁sensor的校准数据及校准精度，使用函数参数mag_raw_data */
    static float cali_data[3] = {0.0f, 0.0f, 0.0f};
    static float mag_data[3] = {0.0f, 0.0f, 0.0f};
    static int cali_accuracy = 0;
```

```
mag_data[0] = mag_raw_data[0];
mag_data[1] = mag_raw_data[1];
mag_data[2] = mag_raw_data[2];
//调用地磁sensor厂商的算法库接口calibrate_alg(), 使用时需包含该接口头文件
calibrate_alg(mag_raw_data, &(cali_data[0]), &cali_accuracy,...);

*cali_mag = cali_data[0];
*(cali_mag + 1) = cali_data[1];
*(cali_mag + 2) = cali_data[2];
*mag_accuracy = cali_accuracy;

    return 0;
}
/* PatchTableMagClose()根据需实现, 函数参数根据实际需要使⽤*/
static int PatchTableMagClose(float *mag_offset)
{
    /* disable时调用, 获取mag sensor当前的offset等参数*/
    //例如获取mag sensor当校准精度为3时的X/Y/Z轴的offset值
    if (cali_accuracy == 3) {
        *mag_offset = cali_data[0] - mag_data[0];
        *(mag_offset + 1) = cali_data[1] - mag_data[1];
        *(mag_offset + 2) = cali_data[2] - mag_data[2];
    }

    return 0;
}
/* PatchTableMagUpdateCfg()根据需实现, 函数参数根据实际需要使⽤*/
static int PatchTableMagUpdateCfg(int *arg)
{
    /* enable时调用, 函数传入地磁sensor的软磁校准参数, 调用地磁sensor厂商的算法库接口进⾏软磁校准, 使用时需包含该
    算法库接口头文件*/
    //若地磁sensor厂商无需软磁校准, 亦可不做任何操作

    return 0;
}

#pragma arm section rdata = "extern_call"
struct patch_table_sensor extern_sensor_call = {
    PatchTableMagInit,
    PatchTableMagUpdate,
    PatchTableMagClose,
```

Unisoc Confidential For hiar