# iio子系统框架

iio子系统旨在支持ADC、DAC相关设备，陀螺仪、加速度计、光照传感器等内部含有ADC的设备都包含其中。hwmon主要针对低采样率传感器的应用，如风扇控制或温度测量，input则专注于输入设备，如键盘、鼠标、触摸屏等。想象一下一个传感器既可以读取其指标数据又可以进行阈值检测然后上报事件，那么hwmon与input子系统都不能同时满足此需求，iio子系统则可以弥补hwmon与input子系统之间的空白，iio又可以像hwmon那样通过sysfs访问，又可以像input那样向应用层报告事件。

sysfs文件系统

iio-buffer

iio-core

iio-event

iio-trigger

iio设备驱动

设备

用户空间

硬件

字符设备接口

内

核

空

间

核心

驱动

iio通过buffer缓冲支持了数据的连续采集，连续数据采集需要通过字符设备接口读取，而通过sysfs也可以实现单次数据的采集。iio的触发可用于触发事件或者触发缓冲，内核已经提供了几种触发方法，但是无论是事件还是缓冲的触发都是可选的，不通过触发依然可以将数据或者事件传递到应用层。

# iio数据结构

## iio设备

内核通过iio\_dev结构来描述一个iio设备：

struct iio\_dev {

int id;

int modes;

int currentmode;

struct device dev;

struct iio\_event\_interface \*event\_interface;

struct iio\_buffer \*buffer;

struct list\_head buffer\_list;

int scan\_bytes;

struct mutex mlock;

const unsigned long \*available\_scan\_masks;

unsigned masklength;

const unsigned long \*active\_scan\_mask;

bool scan\_timestamp;

unsigned scan\_index\_timestamp;

struct iio\_trigger \*trig;

bool trig\_readonly;

struct iio\_poll\_func \*pollfunc;

struct iio\_poll\_func \*pollfunc\_event;

struct iio\_chan\_spec const \*channels;

int num\_channels;

struct list\_head channel\_attr\_list;

struct attribute\_group chan\_attr\_group;

const char \*name;

const struct iio\_info \*info;

clockid\_t clock\_id;

struct mutex info\_exist\_lock;

const struct iio\_buffer\_setup\_ops \*setup\_ops;

struct cdev chrdev;

};

.mode：设备支持的模式，可支持的模式如下：

INDIO\_DIRECT\_MODE：表示可通过sysfs直接读取单次数据。

INDIO\_BUFFER\_TRIGGERED：表示设备支持缓冲触发。

INDIO\_BUFFER\_SOFTWARE：表示支持软件缓冲。

INDIO\_BUFFER\_HARDWARE：表示设备支持硬件缓冲，这三种缓冲方式都是连续采集时的数据缓冲方式。

INDIO\_EVENT\_TRIGGERED：表示支持事件触发，如针对温度传感器可监控当前温度是否超过温度告警上限或下限，当出现温度告警后则向SOC发送中断信号，若支持这类功能则增加INDIO\_EVENT\_TRIGGRED模式的支持即可（事件告警信息也是通过访问字符设备文件读取，但这个字符设备文件有点特殊，其是一个匿名文件）。

.currentmode：当前使用的模式。

.dev：对应的设备，会被绑到iio\_bus上。

.event\_interface：事件相关的数据结构。

.buffer：数据缓冲区。

.buffer\_list：使能数据缓冲区后，会被加入到该链表。

.scan\_bytes：采集并提供给缓冲区的字节数，注意因为在计算scan\_bytes时有对齐操作，比如有两个ad通道，每个通道数据为1个字节，时间戳通道是8个字节，此时scan\_bytes并不为10，而是16，此时应用层读到的也是16个字节，前两个字节为ad数据，从第8个字节开始的8个字节则为时间戳。

.available\_scan\_masks：设备可用通道的掩码，如八通道设备只有前4通道可用时，可将此值设置为0xf。

.active\_scan\_mask：使能通道的掩码，如八通道设备只使能前两个通达，则该值应为0x3，该值为available\_scan\_masks的子集。

.scan\_timestamp：表示通过buffer采集的数据是否需要时间戳，如果为true，则会增加IIO\_TIMESTAMP类型的虚拟通道将时间戳会作为缓冲区的最后一个元素进行推送。时间戳是8字节长。

.scan\_index\_timestamp：时间戳索引。

.trig：设置当前的触发器。

.pollfunc：收到缓冲触发时执行的回调。

.pollfunc\_event：收到事件触发时执行的回调。

.channels：用于描述设备所有的通道信息。

.num\_channels：通道数目。

.channel\_attr\_list：此链表包含了所有通道的属性。

.name：设备名称。

.info：驱动程序需要提供的回调函数与信息。

.setup\_ops：启用/禁用缓冲区前后会被调用的回调函数。

## iio通道

iio设备可能含有多个通道，每个通道通过iio\_chan\_spec结构描述：

struct iio\_chan\_spec {

enum iio\_chan\_type type;

int channel;

int channel2;

unsigned long address;

int scan\_index;

struct {

char sign;

u8 realbits;

u8 storagebits;

u8 shift;

u8 repeat;

enum iio\_endian endianness;

} scan\_type;

long info\_mask\_separate;

long info\_mask\_shared\_by\_type;

long info\_mask\_shared\_by\_dir;

long info\_mask\_shared\_by\_all;

const struct iio\_event\_spec \*event\_spec;

unsigned int num\_event\_specs;

const struct iio\_chan\_spec\_ext\_info \*ext\_info;

const char \*extend\_name;

const char \*datasheet\_name;

unsigned modified:1;

unsigned indexed:1;

unsigned output:1;

unsigned differential:1;

};

.type：通道的测量类型，所有可用类型定义于在include/uapi/linux/iio/types.h，各个类型的含义可参照industrialio-core.c中定义的iio\_chan\_type\_name\_spec就可知测量类型的具体含义，截取部分如下：

static const char \* const iio\_chan\_type\_name\_spec[] = {

[IIO\_VOLTAGE] = "voltage", //电压

[IIO\_CURRENT] = "current", //电流

[IIO\_LIGHT] = "illuminance", //光强

}；

.channel：当.indexed为1时用于指定通道的索引。

.channel2：当.modified为1时用于表示通道的别称，如三轴陀螺仪，用X/Y/Z才好进行区分，可用的 别名定义于include/uapi/linux/iio/types.h的enum iio\_modifier，如三轴陀螺仪的三个通道可 用IIO\_MOD\_X、IIO\_MOD\_Y、IIO\_MOD\_Z分别表示，其最终转换的名称在industrialio-core.c的 iio\_modifier\_names中定义，如[IIO\_MOD\_X] = "x"。

.address：驱动自定义的标识，如可赋值为寄存器地址等，在回调函数中使用。

.scan\_index：当使用缓冲时指示了通道在缓冲区中的位置，如果为-1则表明该通道数据不会进入缓冲区。

.scan\_type：当使用缓冲时表示数据类型。

struct {

char sign;//’s’-有符号 ’u’-无符号

u8 realbits;//数据中的有效位数

u8 storagebits;//realbits+填充

u8 shift;//右移位数

u8 repeat;//不知何意

enum iio\_endian endianness;//大小端

} scan\_type;

.info\_mask\_separate：表示通道的专属属性。

.info\_mask\_shared\_by\_type：表示由相同类型即.type相同的通道共享的属性。

.info\_mask\_shared\_by\_dir：表示由相同方向的通道共享的属性。

.info\_mask\_shared\_by\_all：表示由所有通道共享的属性。

通道属性信息在include/linux/iio/iio.h中使用enum iio\_chan\_info\_enum 定义，其含义可

参考industrialio-core.c的iio\_chan\_info\_postfix。注意以上属性赋值都是按位表示，所以赋值 形式应该使用BIT()，如：.info\_mask\_separate = BIT(IIO\_CHAN\_INFO\_PROCESSED)。

.event\_spec：通道的事件信息，在iio事件小结中详细介绍。

.num\_event\_specs：事件的个数。

.modified：指示通道是否使用.channel2中定义的别称。

.indexed：指示通道是否具有索引，有才会使用.channel中指定的索引。

.output：通道方向，1则为输出，0则为输入。

按照iio通道配置会在/sys/bus/iio/devices/iio:deviceX/下创建通道属性文件用于读取通道数据，属性文件的名称规则为：

{direction}\_{type}\_{index}\_{modifier}\_{info\_mask}

## iio访问

iio\_info结构定义了iio核心访问iio设备的回调，通过sysfs读写通道属性时会执行这些回调：

struct iio\_info {

struct module \*driver\_module;//一般为THIS\_MODULE。

struct attribute\_group \*event\_attrs;//自定义事件属性

const struct attribute\_group \*attrs;//自定义设备属性

int (\*read\_raw)(struct iio\_dev \*indio\_dev, struct iio\_chan\_spec const \*chan, int \*val, int \*val2, long mask);

int (\*read\_raw\_multi)(struct iio\_dev \*indio\_dev, struct iio\_chan\_spec const \*chan, int max\_len, int \*vals, int \*val\_len, long mask);

int (\*write\_raw)(struct iio\_dev \*indio\_dev, struct iio\_chan\_spec const \*chan, int val, int val2, long mask);

int (\*write\_raw\_get\_fmt)(struct iio\_dev \*indio\_dev, struct iio\_chan\_spec const \*chan, long mask);

int (\*read\_event\_config)(struct iio\_dev \*indio\_dev, const struct iio\_chan\_spec \*chan,enum iio\_event\_type type, enum iio\_event\_direction dir);

int (\*write\_event\_config)(struct iio\_dev \*indio\_dev, const struct iio\_chan\_spec \*chan, enum iio\_event\_type type, enum iio\_event\_direction dir, int state);

int (\*read\_event\_value)(struct iio\_dev \*indio\_dev,const struct iio\_chan\_spec \*chan,enum iio\_event\_type type,enum iio\_event\_direction dir,enum iio\_event\_info info, int \*val, int \*val2);

int (\*write\_event\_value)(struct iio\_dev \*indio\_dev,const struct iio\_chan\_spec

\*chan,enum iio\_event\_type type,enum iio\_event\_direction dir,enum iio\_event\_info info, int val, int val2);

int (\*validate\_trigger)(struct iio\_dev \*indio\_dev,struct iio\_trigger \*trig);

int (\*update\_scan\_mode)(struct iio\_dev \*indio\_dev,const unsigned long \*scan\_mask);

int (\*debugfs\_reg\_access)(struct iio\_dev \*indio\_dev,

unsigned reg, unsigned writeval,

unsigned \*readval);

int (\*of\_xlate)(struct iio\_dev \*indio\_dev,

const struct of\_phandle\_args \*iiospec);

int (\*hwfifo\_set\_watermark)(struct iio\_dev \*indio\_dev, unsigned val);

int (\*hwfifo\_flush\_to\_buffer)(struct iio\_dev \*indio\_dev,

unsigned count);

};

.read\_raw：读取通道数据，参数说明如下：

.mask：所请求数据的信息，为iio\_chan\_spec->info\_mask中设置的通道信息。

.val/.val2：构成返回值的元素。

返回值：不同返回值决定了val和val2的使用方式，代码实现在industrialio-core.c中的iio\_format\_value(),如果val=1，val2=2，则在如下不同返回值下sysfs看到的结果如下：

IIO\_VAL\_INT：1

IIO\_VAL\_INT\_PLUS\_MICRO：1.000002

IIO\_VAL\_INT\_PLUS\_NANO：1.000000002

IIO\_VAL\_INT\_PLUS\_MICRO\_DB：1.000002 dB

IIO\_VAL\_INT\_MULTIPLE：1 2

IIO\_VAL\_FRACTIONAL：0.500000000 (val/val2)

IIO\_VAL\_FRACTIONAL\_LOG2：0.250000000 (val>>val2)

.read\_raw\_multi：优先级比read\_raw()高，如果定义此回调则在读取通道数据时优先执行此回调，可用于读取val\_len个值到vals，mask和返回值含义同上。

.write\_raw：将数据写入对应通道。

.mask：同上。

.val/val2：分别为写入数据的整数部分小数部分，如果未定义write\_raw\_get\_fmt回调，如果将 1.2 写入属性文件，则得到的val以及val2分别为1和200000，小数部分会乘以100000。

.write\_raw\_get\_fmt：返回mask对应数据信息的数据类型，决定了write\_raw()回调中得到的val与val2值，如果将1.2写入属性文件，则在不同返回值下val和val2值分别为：

IIO\_VAL\_INT：写入会失败，因为只支持整形。

IIO\_VAL\_INT\_PLUS\_MICRO：val=1，val2=200000(小数部分会乘以100000)

IIO\_VAL\_INT\_PLUS\_NANO：val=1，val2=200000000(小数部分会乘以100000000)

.read\_event\_config：获取事件是否使能。

.write\_event\_config：设置事件是否使能。

.read\_event\_value：读取事件的当前值，其输入参数iio通道、事件type、事件dir、事件info信息都是为了让我们要读取的是哪个通道的哪个事件的值，与read\_raw()具有一样的返回值，以同样的方式影响着val和val2的组合方式。

.write\_event\_value：写入事件的值，同样其输入参数都是为了让我们要读取的是哪个通道的哪个事件的值，但是此回调的数据类型不能更改，如果将1.2写入事件属性文件，则val和val2分别为1和200000。

## iio事件

iio的每个通道都可以设置多个事件，通道的事件可像input子系统那样向用户层报告事件信息，如有的温度传感器可以设置阈值，如果温度超过阈值可通过gpio电平的变化通知cpu，cpu则可捕捉gpio中断即为事件发生，事件信息使用iio\_event\_spec结构描述：

struct iio\_event\_spec {

enum iio\_event\_type type;

enum iio\_event\_direction dir;

unsigned long mask\_separate;

unsigned long mask\_shared\_by\_type;

unsigned long mask\_shared\_by\_dir;

unsigned long mask\_shared\_by\_all;

};

.type：事件类型，可选事件类型如下。除了阈值类型其它类型是什么含义？。

enum iio\_event\_type {

IIO\_EV\_TYPE\_THRESH,//超出阈值

IIO\_EV\_TYPE\_MAG,

IIO\_EV\_TYPE\_ROC,

IIO\_EV\_TYPE\_THRESH\_ADAPTIVE,

IIO\_EV\_TYPE\_MAG\_ADAPTIVE,

IIO\_EV\_TYPE\_CHANGE,

};

.dir：事件方向，可选方向如下：

enum iio\_event\_direction {

IIO\_EV\_DIR\_EITHER,

IIO\_EV\_DIR\_RISING,

IIO\_EV\_DIR\_FALLING,

IIO\_EV\_DIR\_NONE,

};

.mask\_separate：表示通道的专属事件属性。

.mask\_shared\_by\_type：表示同类型通道共享的事件属性。

.mask\_shared\_by\_dir：表示同类型同方向通道共享的事件属性。

.mask\_shared\_by\_all：表示被所有通道共享的事件属性。

事件属性信息include/linux/iio/types.h中使用enum iio\_event\_info 定义，事件属性的赋

值也是按位赋值。

enum iio\_event\_info {

IIO\_EV\_INFO\_ENABLE,

IIO\_EV\_INFO\_VALUE,

IIO\_EV\_INFO\_HYSTERESIS,

IIO\_EV\_INFO\_PERIOD,

IIO\_EV\_INFO\_HIGH\_PASS\_FILTER\_3DB,

IIO\_EV\_INFO\_LOW\_PASS\_FILTER\_3DB,

};

## iio缓冲

iio\_buffer用于作为连续数据采集的缓存：

struct iio\_buffer {

int length;

int bytes\_per\_datum;

struct attribute\_group \*scan\_el\_attrs;

long \*scan\_mask;

bool scan\_timestamp;

const struct iio\_buffer\_access\_funcs \*access;

struct list\_head scan\_el\_dev\_attr\_list;

struct attribute\_group buffer\_group;

struct attribute\_group scan\_el\_group;

wait\_queue\_head\_t pollq;

bool stufftoread;

const struct attribute \*\*attrs;

struct list\_head demux\_list;

void \*demux\_bounce;

struct list\_head buffer\_list;

struct kref ref;

unsigned int watermark;

};

.length：缓冲数据的个数。

.bytes\_per\_datum：每次采集数据的长度，bytes\_per\_datum\*length就是缓冲区大小。

.scan\_el\_attrs：自定义的属性，会生成在scan\_elements目录下。

.scan\_mask：连续采集时的通道掩码。

.scan\_timestamp：连续采集数据是否包含时间戳，当我们在配置iio通道时添加了时间戳通道，那么在

scan\_elements目录就会出现in\_timestamp\_en，写入1时就会在数据中加入时间戳。

.access：缓冲区的访问函数，下面会详细介绍。

.scan\_el\_dev\_attr\_list：iio buffer的所有属性的列表，用于属性文件重名检查等。

.buffer\_group：iio buffer属性，会在buffer目录下创建对应的属性文件。

.scan\_el\_group：iio buffer属性，会在scan\_elements目录下创建对应的属性文件。

.pollq：等待队列，用于iio字符设备read或者poll时等待buffer中有数。

.attrs：自定义的属性，会生成在buffer目录下。

.watermark：缓存watermark\*bytes\_per\_datum数量数据后才唤醒pollq。

缓冲区的访问由iio\_buffer\_access\_funcs结构表示：

struct iio\_buffer\_access\_funcs {

int (\*store\_to)(struct iio\_buffer \*buffer, const void \*data);

int (\*read\_first\_n)(struct iio\_buffer \*buffer,size\_t n, char \_\_user \*buf);

size\_t (\*data\_available)(struct iio\_buffer \*buffer);

int (\*request\_update)(struct iio\_buffer \*buffer);

int (\*set\_bytes\_per\_datum)(struct iio\_buffer \*buffer, size\_t bpd);

int (\*set\_length)(struct iio\_buffer \*buffer, int length);

int (\*enable)(struct iio\_buffer \*buffer, struct iio\_dev \*indio\_dev);

int (\*disable)(struct iio\_buffer \*buffer, struct iio\_dev \*indio\_dev);

void (\*release)(struct iio\_buffer \*buffer);

unsigned int modes;

unsigned int flags;

};

.store\_to：向buffer中写入数据。

.read\_first\_n：从buffer中读取n个字节的数据。

.set\_length：设置buffer长度。

## iio触发

iio的连续数据采集或者事件可使用触发机制：

struct iio\_trigger {

const struct iio\_trigger\_ops \*ops;

int id;

const char \*name;

struct device dev;

struct list\_head list;

struct list\_head alloc\_list;

atomic\_t use\_count;

struct irq\_chip subirq\_chip;

int subirq\_base;

struct iio\_subirq subirqs[CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER];

unsigned long pool[BITS\_TO\_LONGS(CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER)];

struct mutex pool\_lock;

bool attached\_own\_device;

};

.ops：iio trigger的操作函数，下面详细介绍。

.id：id号。

.name：名称。

.dev：会被加入iio总线的设备。

.list：会被添加到全局的触发器链表iio\_trigger\_list中。

.use\_count：使用计数。

.subirq\_chip/subirq\_base/subirqs/pool/pool\_lock：iio\_trigger会创建虚拟的irq\_chip，然后每个trigger consumer注册时都会被分配虚拟的irq，然后使用consumer提供的pollfunc为该irq注册中断，此后，如果trigger触发后则会遍历所有该trigger上已注册的irq调用其中断处理函数，具体过程在驱动开发iio触发使用小结详解。

iio\_trigger的操作函数如下：

struct iio\_trigger\_ops {

struct module \*owner;

int (\*set\_trigger\_state)(struct iio\_trigger \*trig, bool state);

int (\*try\_reenable)(struct iio\_trigger \*trig);

int (\*validate\_device)(struct iio\_trigger \*trig, struct iio\_dev \*indio\_dev);

};

.owner：一般为THIS\_MODULE。

.set\_trigger\_state：用于开/关trigger。

.try\_reenable：使用计数为零时重新启用触发器，可以为NULL。

.validate\_device：用于当前触发器更改时验证设备。

# iio设备驱动开发

## iio设备注册

iio设备注册前需要提前为iio设备申请内存，该函数不仅申请了iio\_dev还申请了sizeof\_priv大小的额外内存用户驱动存放私有数据：

struct iio\_dev \*devm\_iio\_device\_alloc(struct device \*dev, int sizeof\_priv);

可通过如下函数得到私有数据的地址：

void \*iio\_priv(const struct iio\_dev \*indio\_dev);

如果使用的是iio\_device\_alloc()申请的内存还需要退出时进行释放：

void iio\_device\_free(struct iio\_dev \*dev);

然后就需要根据需要填充iio\_dev的信息，然后注册iio设备：

int devm\_iio\_device\_register(struct device \*dev, struct iio\_dev \*indio\_dev);

如果iio\_device\_register()注册设备还需要退出时注销设备：

void iio\_device\_unregister(struct iio\_dev \*indio\_dev);

## iio事件使用

事件属性参数在/sys/bus/iio/devices/iio:deviceX/events下创建属性文件的名称规则为：

{iio\_dir}\_{iio\_channel\_type}{channel-Index/channel\_modify}\_{ev\_type}\_{ev\_dir}\_{ev\_info}

想要使用iio事件，iio\_dev->iio\_info须提供对事件的读取与配置的回调：

.read\_event\_value：读取与事件关联的配置值。

.write\_event\_value：为事件写入一个配置值。

事件需要通过向/sys/bus/iio/devices/iio:deviceX/events目录下的xxx\_en属性文件写1使能事件通知。

内核在事件发生后需要将事件添加到列表中，以供用户层读取:

int iio\_push\_event(struct iio\_dev \*indio\_dev, u64 ev\_code, s64 timestamp);

.indio\_dev：iio设备。

.ev\_code：事件编码，包括通道类型、方向、事件类型等，内核在include/linux/iio/events.h中提供了宏来创建编码：

IIO\_EVENT\_CODE(chan\_type, diff, modifier, direction, type, chan, chan1, chan2)

IIO\_MOD\_EVENT\_CODE(chan\_type, number, modifier, type, direction)

IIO\_UNMOD\_EVENT\_CODE(chan\_type, number, type, direction)

传递到应用层后，也可利用include/uapi/linux/iio/events.h中的宏解析编码信息：

IIO\_EVENT\_CODE\_EXTRACT\_TYPE(mask)

IIO\_EVENT\_CODE\_EXTRACT\_DIR(mask)

IIO\_EVENT\_CODE\_EXTRACT\_CHAN\_TYPE(mask)

IIO\_EVENT\_CODE\_EXTRACT\_CHAN(mask)

IIO\_EVENT\_CODE\_EXTRACT\_MODIFIER(mask)

.timestamp：事件发生时间，可使用iio\_get\_time\_ns()生成时间戳。

## iio缓冲区使用

要想支持缓冲区，驱动中必须对iio\_chan\_spec的.scan\_index和.scan\_type进行初始化。

常用的iio缓冲区为kfifo缓冲，使用前需要申请，其会帮助我们初始化iio\_buffer并且会提供缓冲区的访问方法iio\_buffer\_access\_funcs：

struct iio\_buffer \*devm\_iio\_kfifo\_allocate(struct device \*dev);

然后需要将缓冲区与iio设备绑定：

iio\_device\_attach\_buffer(struct iio\_dev \*indio\_dev, struct iio\_buffer \*buffer);

然后需要将iio\_dev的模式支持上软件缓冲：

indio\_dev->modes |= INDIO\_BUFFER\_SOFTWARE;

此后就可以在中断处理中或者适当的位置向缓冲区推入输入：

int iio\_push\_to\_buffers\_with\_timestamp(struct iio\_dev \*indio\_dev, void \*data, int64\_t timestamp);

注意：data buffer要足够大来容纳额外的时间戳，通常大小为iio\_dev->scan\_bytes，但是注意比如有两个ad通道，每个通道数据为1个字节，时间戳通道是8个字节，此时scan\_bytes并不为10，而是16，在计算scan\_bytes时会有对齐操作，详情请参考industrialio-buffer.c中iio\_compute\_scan\_bytes()计算scan\_bytes的过程。所以我们在提供data内存空间时也可借用对齐操作，对于两个ad通道，每个通道数据为1个字节，还需传递时间戳的情况，可以使用unsigned char data[ALIGN(2, sizeof(s64)) + sizeof(s64)]来完成内存的申请，当然也可以通过kzalloc(iio\_dev->scan\_bytes, GFP\_KERNEL)。

注意此时就可以在应用层读取数据了，但前提是已经使能了至少一个通道以及使能了buffer:

echo 1 > /sys/bus/iio/devices/iio\:device1/scan\_elements/in\_illuminance\_en

echo 1 > /sys/bus/iio/devices/iio\:device1/buffer/enable

## iio触发使用

### 使用方法

iio trigger的作用就是当某一条件满足时调用每个使用该trigger的consumer的回调函数，iio trigger驱动是独立的，trigger可用于触发iio buffer也可用于iio event。

注册iio trigger前需要申请，输入参数用于设置触发器的名称:

struct iio\_trigger \*iio\_trigger\_alloc(const char \*fmt, ...);

为触发器设置私有数据：

void iio\_trigger\_set\_drvdata(struct iio\_trigger \*trig, void \*data);

注册触发器：

int iio\_trigger\_register(struct iio\_trigger \*trig\_info);

注销触发器：

void iio\_trigger\_unregister(struct iio\_trigger \*trig\_info);

触发器注册完成后，consumer就需要与trigger绑定：

int iio\_trigger\_attach\_poll\_func(struct iio\_trigger \*trig, struct iio\_poll\_func \*pf);

但是上面的绑定函数并不面向驱动开发人员，实际调用此函数的分为两种情况：

一种是触发事件：

当用户层向/sys/bus/iio/devices/iio:device1/trigger/current\_trigger写入触发器时会调用iio\_trigger\_write\_current()：

static ssize\_t iio\_trigger\_write\_current(struct device \*dev, struct device\_attribute \*attr, const char \*buf, size\_t len)

{

iio\_trigger\_get(indio\_dev->trig);

if (indio\_dev->modes & INDIO\_EVENT\_TRIGGERED)//判断是否支持事件触发

iio\_trigger\_attach\_poll\_func(indio\_dev->trig, indio\_dev->pollfunc\_event);

}

另一种则是触发缓冲:

当用户层向/sys/bus/iio/devices/iio\:device1/buffer/enable写1使能缓冲时：

static ssize\_t iio\_buffer\_store\_enable(struct device \*dev, struct device\_attribute \*attr, const char \*buf, size\_t len)

{

ret = \_\_iio\_update\_buffers(indio\_dev, indio\_dev->buffer, NULL);

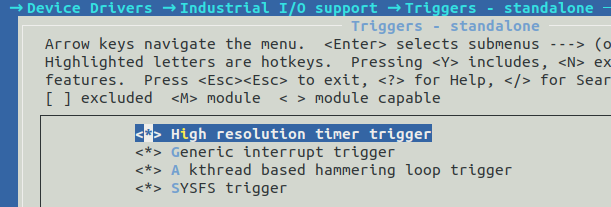
ret = iio\_enable\_buffers(indio\_dev, &new\_config);

ret = indio\_dev->setup\_ops->postenable(indio\_dev);

}

而postenable在iio\_triggered\_buffer\_setup()时被赋值为iio\_triggered\_buffer\_postenable()，其内部就调用了iio\_trigger\_attach\_poll\_func()。

一般情况下我们无需自己实现触发，常用的触发方式内核已经实现，包括通过外部中断、定时器中断以及写特定sysfs文件，将需要的触发方式编入内核即可使用：



### 实现分析

触发器的实现使用了虚拟的irq chip，其在iio\_trigger\_alloc()中进行初始化：

struct iio\_trigger \*iio\_trigger\_alloc(const char \*fmt, ...)

{

trig = kzalloc(sizeof \*trig, GFP\_KERNEL);

trig->subirq\_base = irq\_alloc\_descs(-1, 0, CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER, 0);

for (i = 0; i < CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER; i++) {

irq\_set\_chip(trig->subirq\_base + i, &trig->subirq\_chip);

irq\_set\_handler(trig->subirq\_base + i, &handle\_simple\_irq);

irq\_modify\_status(trig->subirq\_base + i, IRQ\_NOREQUEST | IRQ\_NOAUTOEN, IRQ\_NOPROBE);

}

}

当trigger consumer与trigger绑定时：

static int iio\_trigger\_attach\_poll\_func(struct iio\_trigger \*trig, struct iio\_poll\_func \*pf)

{

//申请中断号

pf->irq = iio\_trigger\_get\_irq(trig);

//注册中断处理函数，中断的上半部和下半部都来自于iio\_poll\_func

ret = request\_threaded\_irq(pf->irq, pf->h, pf->thread,pf->type, pf->name, pf);

}

而事件或者缓冲就是trigger consumer，上面的iio\_poll\_func在触发事件时为iio\_dev->pollfunc\_event，触发缓冲时为iio\_dev->pollfunc，初始化工作可使用如下函数：

struct iio\_poll\_func \*iio\_alloc\_pollfunc(irqreturn\_t (\*h)(int irq, void \*p), irqreturn\_t (\*thread)(int irq, void \*p), int type, struct iio\_dev \*indio\_dev, const char \*fmt, ...);

.h：中断上半部。

.thread：中断下半部。

之后触发发生时，iio\_trigger\_poll()函数会被调用，比如通过写sysfs触发时：

static ssize\_t iio\_sysfs\_trigger\_poll(struct device \*dev, struct device\_attribute \*attr, const char \*buf, size\_t count)

{

iio\_trigger\_poll(trig->trig);

}

void iio\_trigger\_poll(struct iio\_trigger \*trig)

{

if (!atomic\_read(&trig->use\_count)) {

// CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER为一个trigger支持的consumer个数，可通过配 置内核更改，默认为2

atomic\_set(&trig->use\_count, CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER);

for (i = 0; i < CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER; i++) {

if (trig->subirqs[i].enabled)

//如果中断使能了，则回调被执行

generic\_handle\_irq(trig->subirq\_base + i);

else

//否则执行此函数

iio\_trigger\_notify\_done(trig);

}

}

}

void iio\_trigger\_notify\_done(struct iio\_trigger \*trig)

{

//减少trig的引用计数

if (atomic\_dec\_and\_test(&trig->use\_count) && trig->ops->try\_reenable)

if (trig->ops->try\_reenable(trig))

iio\_trigger\_poll(trig);

}

应该注意到iio\_trigger\_poll()中使能了中断的consumer的引用计数并未被减掉，如果不减掉下次再调用iio\_trigger\_poll()时atomic\_read(&trig->use\_count)不为0则回调不会再执行，这也是为了保证下次触发时上次触发的工作已全部完成，所以被使能的consumer的引用计数需要驱动开发者自己减少，只需在中断下半部中结束时调用iio\_trigger\_notify\_done()即可。

### 使用sysfs触发

使用sysfs触发首先需要创建触发器，0为分配给触发器的索引，执行完毕后会在/sys/bus/iio/devices/下生成trigger0触发器目录，其name属性文件为触发器名称是唯一的标识：

echo 0 > /sys/bus/iio/devices/iio\_sysfs\_trigger/add\_trigger

触发器创建完毕后可分配给设备：

cat /sys/bus/iio/devices/trigger0/name >/sys/bus/iio/devices/iio:device1/trigger/current\_trigger

启用扫描元素，选择需要的通道以使通道数据推入缓冲区：

echo 1 > /sys/bus/iio/devices/iio\:device1/scan\_elements/in\_illuminance\_en

设置缓冲区大小：

echo 10 > /sys/bus/iio/devices/iio\:device1/buffer/length

启用缓冲区：

echo 1 > /sys/bus/iio/devices/iio\:device1/buffer/enable

触发触发器以采集数据：

echo 1 > /sys/bus/iio/devices/trigger0/trigger\_now

使用完毕后需要禁用缓冲区以及分离触发器：

echo 0 > /sys/bus/iio/devices/iio\:device1/buffer/enable

echo "" > /sys/bus/iio/devices/iio:device1/trigger/current\_trigger

## iio缓冲触发使用

如果想通过触发向缓冲区写数据，需要在注册iio\_dev前设置触发缓冲区以及触发的回调函数：

int iio\_triggered\_buffer\_setup(struct iio\_dev \*indio\_dev, irqreturn\_t (\*h)(int irq,

void \*p), irqreturn\_t (\*thread)(int irq, void \*p), const struct iio\_buffer\_setup\_ops \*setup\_ops)

{

//使用kfifo作为缓冲区

buffer = iio\_kfifo\_allocate();

//将缓冲区与iio设备绑定

iio\_device\_attach\_buffer(indio\_dev, buffer);

//初始化回调

indio\_dev->pollfunc = iio\_alloc\_pollfunc(h, thread, IRQF\_ONESHOT, indio\_dev, "%s\_consumer%d", indio\_dev->name, indio\_dev->id);

}

触发的回调函数中需要调用iio\_push\_to\_buffers\_with\_timestamp()向buffer中写数，此后就可以在应用层使能缓冲区，然后通过触发触发器向缓冲区写入数据了。

## iio事件触发使用

如果想通过触发完成事件上报，需要在iio\_dev前设置触发的回调函数：

int iio\_triggered\_event\_setup(struct iio\_dev \*indio\_dev, irqreturn\_t (\*h)(int irq,

void \*p), irqreturn\_t (\*thread)(int irq, void \*p))

{

//初始化回调

indio\_dev->pollfunc = iio\_alloc\_pollfunc(h, thread, IRQF\_ONESHOT, indio\_dev, "%s\_consumer%d", indio\_dev->name, indio\_dev->id);

indio\_dev->modes |= INDIO\_EVENT\_TRIGGERED;

}

触发的回调函数中需要调用iio\_push\_event()完成事件的上报，此后就可以在应用层通过触发触发器完成事件的上报。

# iio工具

内核目录tools/iio下提供了很多工具来调试iio，我们可以借用或参照这些工具来加速我们的开发：

lsiio：列出IIO触发器、设备以及可访问的通道。

iio\_event\_monitor：监控IIO设备的IIO事件。

iio\_generic\_buffer：监控、处理、打印从IIO设备缓冲区里接收到的数据。

Analog开发了用于iio设备交互的库：

<https://github.com/analogdevicesinc/libiio>

## iio事件监测

用户层想要监测iio时间，并不能直接读取iio设备的fd，需要通过设备/dev/iio:deviceX的ioctl接口来得到事件的fd：

ioctl(fd, IIO\_GET\_EVENT\_FD\_IOCTL, &event\_fd);

此后就可以进行事件读取了：

read(event\_fd, &event, sizeof(event));

# 示例说明

示例虚拟两个单通道的光传感器，每个通道都支持设置过低告警阈值以及过高告警阈值。

示例中通过定时器模拟光照度的变化，通过fake\_iiodev\_write\_event\_value()可以设置低告警阈值或高告警阈值，再通过fake\_iiodev\_write\_event\_config()使能事件，此后就可在应用层监测事件了。

对于缓冲区的使用，用到了两种方式：软件缓冲和触发缓冲，通过代码中IIO\_BUFFER\_TRIGGERED宏选择。

设备树中需要添加如下节点：

fake\_iiodev: fake\_iiodev@1 {

compatible = "xm,fake\_iiodev";

reg = <0x1>;

};

测试程序中创建了三个线程分别用于监测event、监测buffer以及完成触发，触发使用的是sysfs触发。