目录

[查看文档： 1](#_Toc465269465)

[/dev /sys/dev 及sys/devices之间的关系 3](#_Toc465269466)

[Sensor stack 4](#_Toc465269467)

[SDK 4](#_Toc465269468)

[Framework 5](#_Toc465269469)

[Impact of multiplexing 5](#_Toc465269470)

[Sensor fusion 6](#_Toc465269471)

[Under the Hood 6](#_Toc465269472)

[HAL 7](#_Toc465269473)

[Release cycle 7](#_Toc465269474)

[Kernel driver 7](#_Toc465269475)

[Sensor hub 8](#_Toc465269476)

[Sensors 8](#_Toc465269477)

[Reporting modes 9](#_Toc465269478)

[IN THIS DOCUMENT 9](#_Toc465269479)

[Continuous 9](#_Toc465269480)

[On-change 9](#_Toc465269481)

[One-shot 10](#_Toc465269482)

[Special 10](#_Toc465269483)

[Suspend mode 10](#_Toc465269484)

[IN THIS DOCUMENT 10](#_Toc465269485)

[SoC power states 11](#_Toc465269486)

[Non-wake-up sensors 11](#_Toc465269487)

[Wake-up sensors 11](#_Toc465269488)

[How to define wake-up and non-wake-up sensors? 12](#_Toc465269489)

[Power consumption 12](#_Toc465269490)

[IN THIS DOCUMENT 12](#_Toc465269491)

[Low-power sensors 13](#_Toc465269492)

[ Power measurement process 13](#_Toc465269493)

[Interaction 14](#_Toc465269494)

[HAL interface 15](#_Toc465269495)

[get\_sensors\_list(list) 15](#_Toc465269496)

[activate(sensor, true/false) 16](#_Toc465269497)

[batch(sensor, flags, sampling period, maximum report latency) 17](#_Toc465269498)

[sampling\_period\_ns 17](#_Toc465269499)

[max\_report\_latency\_ns 18](#_Toc465269500)

[setDelay(sensor, sampling period) 19](#_Toc465269501)

[flush(sensor) 19](#_Toc465269502)

[poll() 20](#_Toc465269503)

[Sequence of calls 20](#_Toc465269504)

[sensors\_module\_t 21](#_Toc465269505)

[sensors\_poll\_device\_t / sensors\_poll\_device\_1\_t 21](#_Toc465269506)

[sensor\_t 21](#_Toc465269507)

[sensors\_event\_t 23](#_Toc465269508)

[Metadata flush complete events 24](#_Toc465269509)

[Batching 25](#_Toc465269510)

[What is batching? 25](#_Toc465269511)

[Wake-up FIFOs and non-wake-up FIFOs 25](#_Toc465269512)

[Behavior outside of suspend mode 26](#_Toc465269513)

[Behavior in suspend mode 26](#_Toc465269514)

[Precautions to take when batching wake-up sensors 27](#_Toc465269515)

[Precautions to take when batching non-wake-up on-change sensors 27](#_Toc465269516)

[Implementing batching 28](#_Toc465269517)

[FIFO allocation priority 29](#_Toc465269518)

[High value: Low power pedestrian dead reckoning 29](#_Toc465269519)

[High value: Medium power intermittent activity/gesture recognition 29](#_Toc465269520)

[Medium value: Medium power continuous activity/gesture recognition 29](#_Toc465269521)

[Medium-high value: Interrupt load reduction 30](#_Toc465269522)

[Medium value: Continuous low frequency data collection 30](#_Toc465269523)

[Medium-low value: Continuous full-sensors collection 30](#_Toc465269524)

[Base sensors 30](#_Toc465269525)

[Accelerometer 31](#_Toc465269526)

[Ambient temperature 32](#_Toc465269527)

[Magnetic field sensor 32](#_Toc465269528)

[Gyroscope 33](#_Toc465269529)

[Heart Rate 34](#_Toc465269530)

[Light 34](#_Toc465269531)

[Proximity 34](#_Toc465269532)

[Pressure 35](#_Toc465269533)

[Relative humidity 35](#_Toc465269534)

[Composite sensor types 35](#_Toc465269535)

## 查看文档：

<http://blog.csdn.net/luoshengyang/article/details/6567257> sensor

<http://blog.csdn.net/myarrow/article/details/9856095/> selinux

<http://wenku.baidu.com/link?url=GOzsu2gQJn2BPD4XsTBzisQLDhEZuW0JLWbE1qukoWOErNUduUsC8AaCTTsAm5wJwZlJNl-vPuXnPeQtS2fk_277qAwF9A_Mtm-nH1KK8Sa> 百度文档

<http://wenku.baidu.com/view/76af5151a6c30c2258019e00.html?re=view> mtksensor框架

<http://blog.csdn.net/feitian_666/article/details/51612395> sensor

<http://www.cnblogs.com/lcw/p/3402770.html> 这个原版

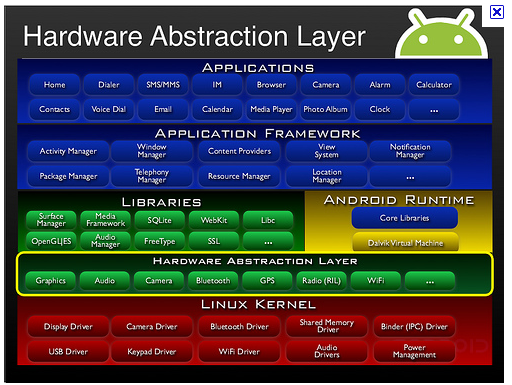
http://www.open-open.com/lib/view/open1325474381453.html

<http://www.cnblogs.com/happy-leon/p/5655614.html>

<http://www.xuebuyuan.com/2128448.html>

http://blog.csdn.net/loongembedded/article/details/51442241

https://source.android.com/devices/sensors/hal-interface.html



struct hw\_module\_t; //模块类型

struct hw\_module\_methods\_t; //模块方法

struct hw\_device\_t; //设备类型

Android中Sensor的HAL接口定义在：hardware/libhardware/include/hardware/sensors.h

#define SENSOR\_TYPE\_ACCELEROMETER 1 //加速度传感器

#define SENSOR\_TYPE\_MAGNETIC\_FIELD 2 //磁力传感器

#define SENSOR\_TYPE\_ORIENTATION 3 //方向

#define SENSOR\_TYPE\_GYROSCOPE 4 //陀螺仪

#define SENSOR\_TYPE\_LIGHT 5 //环境光照传感器

#define SENSOR\_TYPE\_PRESSURE 6 //压力传感器

#define SENSOR\_TYPE\_TEMPERATURE 7 //温度传感器

#define SENSOR\_TYPE\_PROXIMITY 8 //距离传感器

#define SENSOR\_TYPE\_GRAVITY 9 //重力传感器

#define SENSOR\_TYPE\_LINEAR\_ACCELERATION 10 //线性加速度

#define SENSOR\_TYPE\_ROTATION\_VECTOR 11 //旋转矢量传感器

#define SENSOR\_TYPE\_RELATIVE\_HUMIDITY 12 //湿度传感器

#define SENSOR\_TYPE\_AMBIENT\_TEMPERATURE 13 //温度传感器

TYPE\_TEMPERATURE:温度传感器,新版本中被TYPE\_AMBIENT\_TEMPERATURE替换掉了。

1.TYPE\_ACCELEROMETER：加速度传感器，单位是m/s²，测量应用于设备X、Y、Z轴上的加速度，又叫做G-sensor

2.TYPE\_AMBIENT\_TEMPERATURE：温度传感器，单位是℃，测量返回当前的温度。

3.TYPE\_GRAVITY：重力传感器，单位是m/s²，测量应用于设备X、Y、Z轴上的重力，也叫GV-sensor，地球上的数值是9.8m/s²，也可以设置其他星球(呃，目测平时用不到吧)

4.TYPE\_GYROSCOPE：陀螺仪传感器，单位是rad/s，测量设备x、y、z三轴的角加速度数据。

5.TYPE\_LIGHT：光线感应传感器，单位lx，检测周围的光线强度，手机系统中主要是调节LCD亮度。

6.TYPE\_LINEAR\_ACCELERATION：线性加速度传感器，单位是m/s²，该传感器是获取加速度传感器去除重力的影响得到的数据。

7.TYPE\_MAGNETIC\_FIELD：磁力传感器，单位是uT(微特斯拉)，测量设备周围三个物理轴（x，y，z）的磁场(个人不了解哪些App上有应用)。

8.TYPE\_ORIENTATION:方向传感器,测量设备围绕三个物理轴（x，y，z）的旋转角度,API显示使用 SensorManager.getOrientation()替代掉了。

9.TYPE\_PRESSURE：压力传感器，单位是hPa(百帕斯卡)，返回当前环境下的压强。

10.TYPE\_PROXIMITY：距离传感器，单位是cm，用来测量某个对象到屏幕的距离，可用于打电话时判断人耳到电话屏幕距离来进行关闭屏幕的省电功能。

11.TYPE\_RELATIVE\_HUMIDITY：湿度传感器，单位是%，来测量周围环境的相对湿度(估计很少设备有吧)。

标轴和角度混合计算得到的数据(具体咋用不了解)。

13.TYPE\_TEMPERATURE:温度传感器,新版本中被TYPE\_AMBIENT\_TEMPERATURE替换掉了。

## /dev /sys/dev 及sys/devices之间的关系

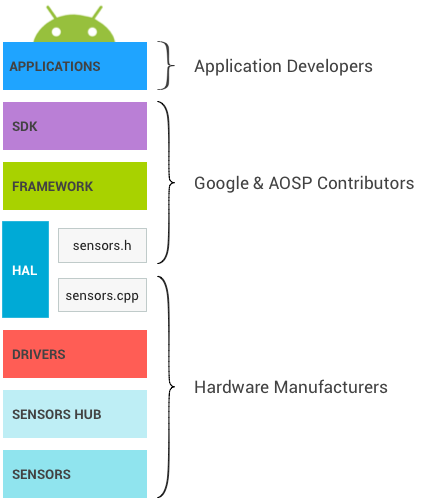
<http://bbs.csdn.net/topics/380166008>

<http://bbs.csdn.net/topics/380166267>

1. sysfs的挂载点是/sys目录， sysfs是一个虚拟的文件系统(还有其它的虚拟文件系统，例如usbfs和procfs)，sysfs导出了内核的数据结构。  
   2. /sys/dev/ 和/sys/devices是sysfs按面向对象管理的思想来组织，sysfs最主要是用来描绘Linux kernel 2.6中的设备驱动模型，用户态的  
   mdev/udev后台程序会动态地周期性的扫描/sys目录中的属性项来自动管理设备文件（也称为设备节点），从而在/dev目录会建立或者删除对应的设备文件。
2. 1、/dev 下放的是设备文件，是由应用层mknod创建的文件。如果底层驱动对mknod的设备号有对应的驱动，如open等函数，那么应用层open "/dev/\*\*"时，就会调用到底层的驱动。说白了，/dev下放的是内核和应用层交互的文件，让应用层去open,write,poll等。  
   2、/sys 是个文件系统，你写内核代码时，如果有调用kobj\_init等函数，就会在/sys下的相应目录生成相应文件。 它的作用是将内核注册的设备、驱动、BUS连成一个树形结构。 另外，应用层也可以通过读写/sys下的文件和内核进行交互（ktype）。 说白了/sys就是一个树形结构，让你明白内核都有哪些驱动和设备已经bus，方便电源管理。

# Sensor stack

The figure below represents the Android sensor stack. Each component communicates only with the components directly above and below it, though some sensors can bypass the sensor hub when it is present. Control flows from the applications down to the sensors, and data flows from the sensors up to the applications.



## SDK

Applications access sensors through the [Sensors SDK (Software Development Kit) API](http://developer.android.com/reference/android/hardware/SensorManager.html). The SDK contains functions to list available sensors and to register to a sensor.

When registering to a sensor, the application specifies its preferred sampling frequency and its latency requirements.

* For example, an application might register to the default accelerometer, requesting events at 100Hz, and allowing events to be reported with a 1-second latency.
* The application will receive events from the accelerometer at a rate of at least 100Hz, and possibly delayed up to 1 second.

See the [developer documentation](https://source.android.com/devices/sensors/index.html#targeted_at_developers) for more information on the SDK.

## Framework

The framework is in charge of linking the several applications to the [HAL](https://source.android.com/devices/sensors/hal-interface.html). The HAL itself is single-client. Without this multiplexing happening at the framework level, only a single application could access each sensor at any given time.

* When a first application registers to a sensor, the framework sends a request to the HAL to activate the sensor.
* When additional applications register to the same sensor, the framework takes into account requirements from each application and sends the updated requested parameters to the HAL.
  + The [sampling frequency](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns) will be the maximum of the requested sampling frequencies, meaning some applications will receive events at a frequency higher than the one they requested.
  + The [maximum reporting latency](https://source.android.com/devices/sensors/hal-interface.html#max_report_latency_ns) will be the minimum of the requested ones. If one application requests one sensor with a maximum reporting latency of 0, all applications will receive the events from this sensor in continuous mode even if some requested the sensor with a non-zero maximum reporting latency. See [Batching](https://source.android.com/devices/sensors/batching.html) for more details.
* When the last application registered to one sensor unregisters from it, the frameworks sends a request to the HAL to deactivate the sensor so power is not consumed unnecessarily.

### Impact of multiplexing

This need for a multiplexing layer in the framework explains some design decisions.

* When an application requests a specific sampling frequency, there is no guarantee that events won’t arrive at a faster rate. If another application requested the same sensor at a faster rate, the first application will also receive them at the fast rate.
* The same lack of guarantee applies to the requested maximum reporting latency: applications might receive events with much less latency than they requested.
* Besides sampling frequency and maximum reporting latency, applications cannot configure sensor parameters.
  + For example, imagine a physical sensor that can function both in “high accuracy” and “low power” modes.
  + Only one of those two modes can be used on an Android device, because otherwise, an application could request the high accuracy mode, and another one a low power mode; there would be no way for the framework to satisfy both applications. The framework must always be able to satisfy all its clients, so this is not an option.
* There is no mechanism to send data down from the applications to the sensors or their drivers. This ensures one application cannot modify the behavior of the sensors, breaking other applications.

### Sensor fusion

The Android framework provides a default implementation for some composite sensors. When a [gyroscope](https://source.android.com/devices/sensors/sensor-types.html#gyroscope), an [accelerometer](https://source.android.com/devices/sensors/sensor-types.html#accelerometer) and a [magnetometer](https://source.android.com/devices/sensors/sensor-types.html#magnetic_field_sensor) are present on a device, but no [rotation vector](https://source.android.com/devices/sensors/sensor-types.html#rotation_vector), [gravity](https://source.android.com/devices/sensors/sensor-types.html#gravity) and [linear acceleration](https://source.android.com/devices/sensors/sensor-types.html#linear_acceleration) sensors are present, the framework implements those sensors so applications can still use them.

The default implementation does not have access to all the data that other implementations have access to, and it must run on the SoC, so it is not as accurate nor as power efficient as other implementations can be. As much as possible, device manufacturers should define their own fused sensors (rotation vector, gravity and linear acceleration, as well as newer composite sensors like the [game rotation vector](https://source.android.com/devices/sensors/sensor-types.html#game_rotation_vector)) rather than rely on this default implementation. Device manufacturers can also request sensor chip vendors to provide them with an implementation.

The default sensor fusion implementation is not being maintained and might cause devices relying on it to fail CTS.

### Under the Hood

This section is provided as background information for those maintaining the Android Open Source Project (AOSP) framework code. It is not relevant for hardware manufacturers.

#### JNI

The framework uses a Java Native Interface (JNI) associated with [android.hardware](http://developer.android.com/reference/android/hardware/package-summary.html) and located in the frameworks/base/core/jni/ directory. This code calls the lower level native code to obtain access to the sensor hardware.

#### Native framework

The native framework is defined in frameworks/native/ and provides a native equivalent to the [android.hardware](http://developer.android.com/reference/android/hardware/package-summary.html)package. The native framework calls the Binder IPC proxies to obtain access to sensor-specific services.

#### Binder IPC

The Binder IPC proxies facilitate communication over process boundaries.

## HAL

The Sensors Hardware Abstraction Layer (HAL) API is the interface between the hardware drivers and the Android framework. It consists of one HAL interface sensors.h and one HAL implementation we refer to as sensors.cpp.

The interface is defined by Android and AOSP contributors, and the implementation is provided by the manufacturer of the device.

The sensor HAL interface is located in hardware/libhardware/include/hardware. See [sensors.h](https://source.android.com/devices/halref/sensors_8h.html) for additional details.

### Release cycle

The HAL implementation specifies what version of the HAL interface it implements by setting your\_poll\_device.common.version. The existing HAL interface versions are defined in sensors.h, and functionality is tied to those versions.

The Android framework currently supports versions 1.0 and 1.3, but 1.0 will soon not be supported anymore. This documentation describes the behavior of version 1.3, to which all devices should upgrade. For details on how to upgrade to 1.3, see [HAL version deprecation](https://source.android.com/devices/sensors/versioning.html).

## Kernel driver

The sensor drivers interact with the physical devices. In some cases, the HAL implementation and the drivers are the same software entity. In other cases, the hardware integrator requests sensor chip manufacturers to provide the drivers, but they are the ones writing the HAL implementation.

In all cases, HAL implementation and kernel drivers are the responsibility of the hardware manufacturers, and Android does not provide preferred approaches to write them.

## Sensor hub

The sensor stack of a device can optionally include a sensor hub, useful to perform some low-level computation at low power while the SoC can be in a suspend mode. For example, step counting or sensor fusion can be performed on those chips. It is also a good place to implement sensor batching, adding hardware FIFOs for the sensor events. See [Batching](https://source.android.com/devices/sensors/batching.html) for more information.

How the sensor hub is materialized depends on the architecture. It is sometimes a separate chip, and sometimes included on the same chip as the SoC. Important characteristics of the sensor hub is that it should contain sufficient memory for batching and consume very little power to enable implementation of the low power Android sensors. Some sensor hubs contain a microcontroller for generic computation, and hardware accelerators to enable very low power computation for low power sensors.

How the sensor hub is architectured and how it communicates with the sensors and the SoC (I2C bus, SPI bus, …) is not specified by Android, but it should aim at minimizing overall power use.

One option that appears to have a significant impact on implementation simplicity is having two interrupt lines going from the sensor hub to the SoC: one for wake-up interrupts (for wake-up sensors), and the other for non-wake-up interrupts (for non-wake-up sensors).

## Sensors

Those are the physical MEMs chips making the measurements. In many cases, several physical sensors are present on the same chip. For example, some chips include an accelerometer, a gyroscope and a magnetometer. (Such chips are often called 9-axis chips, as each sensor provides data over 3 axes.)

Some of those chips also contain some logic to perform usual computations such as motion detection, step detection and 9-axis sensor fusion.

Although the CDD power and accuracy requirements and recommendations target the Android sensor and not the physical sensors, those requirements impact the choice of physical sensors. For example, the accuracy requirement on the game rotation vector has implications on the required accuracy for the physical gyroscope. It is up to the device manufacturer to derive the requirements for physical sensors.

# Reporting modes

## IN THIS DOCUMENT

1. [Continuous](https://source.android.com/devices/sensors/report-modes.html#continuous)
2. [On-change](https://source.android.com/devices/sensors/report-modes.html#on-change)
3. [One-shot](https://source.android.com/devices/sensors/report-modes.html#one-shot)
4. [Special](https://source.android.com/devices/sensors/report-modes.html#special)

Sensors can generate events in different ways called reporting modes; each sensor type has one and only one reporting mode associated with it. Four reporting modes exist.

## Continuous

Events are generated at a constant rate defined by the [sampling\_period\_ns](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns) parameter passed to the batch function. Example sensors using the continuous reporting mode are [accelerometers](https://source.android.com/devices/sensors/sensor-types.html#accelerometer) and [gyroscopes](https://source.android.com/devices/sensors/sensor-types.html#gyroscope).

## On-change

Events are generated only if the measured values have changed. Activating the sensor at the HAL level (calling activate(..., enable=1) on it) also triggers an event, meaning the HAL must return an event immediately when an on-change sensor is activated. Example sensors using the on-change reporting mode are the step counter, proximity, and heart rate sensor types.

The [sampling\_period\_ns](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns) parameter passed to the batch function is used to set the minimum time between consecutive events, meaning an event should not be generated until sampling\_period\_ns nanoseconds elapsed since the last event, even if the value changed since then. If the value changed, an event must be generated as soon as sampling\_period\_ns has elapsed since the last event.

For example, suppose:

* We activate the step counter with sampling\_period\_ns = 10 \* 10^9 (10 seconds).
* And walk for 55 seconds, then stand still for one minute.
* Then the events will be generated about every 10 seconds during the first minute (including at time t=0 because of the activation of the sensor, and t=60 seconds), for a total of seven events, and no event will be generated in the second minute because the value of the step count didn’t change after t=60 seconds.

## One-shot

Upon detection of an event, the sensor deactivates itself and then sends a single event through the HAL. Order matters to avoid race conditions. (The sensor must be deactivated before the event is reported through the HAL). No other event is sent until the sensor is reactivated. [Significant motion](https://source.android.com/devices/sensors/sensor-types.html#significant_motion) is an example of this kind of sensor.

One-shot sensors are sometimes referred to as trigger sensors.

The sampling\_period\_ns and max\_report\_latency\_ns parameters passed to the batch function are ignored. Events from one-shot events cannot be stored in hardware FIFOs; the events must be reported as soon as they are generated.

## Special

See the individual [sensor type descriptions](https://source.android.com/devices/sensors/sensor-types.html) for details on when the events are generated.

# Suspend mode

## IN THIS DOCUMENT

1. [SoC power states](https://source.android.com/devices/sensors/suspend-mode.html#soc_power_states)
2. [Non-wake-up sensors](https://source.android.com/devices/sensors/suspend-mode.html#non-wake-up_sensors)
3. [Wake-up sensors](https://source.android.com/devices/sensors/suspend-mode.html#wake-up_sensors)
4. [How to define wake-up and non-wake-up sensors?](https://source.android.com/devices/sensors/suspend-mode.html#how_to_define_wake-up_and_non-wake-up_sensors)

## SoC power states

The power states of the system on a chip (SoC) are: on, idle, and suspend. “On” is when the SoC is running. “Idle” is a medium power mode where the SoC is powered but doesn't perform any tasks. “Suspend” is a low-power mode where the SoC is not powered. The power consumption of the device in this mode is usually 100 times less than in the “On” mode.

## Non-wake-up sensors

Non-wake-up sensors are sensors that do not prevent the SoC from going into suspend mode and do not wake the SoC up to report data. In particular, the drivers are not allowed to hold wake-locks. It is the responsibility of applications to keep a partial wake lock should they wish to receive events from non-wake-up sensors while the screen is off. While the SoC is in suspend mode, the sensors must continue to function and generate events, which are put in a hardware FIFO. (See [Batching](https://source.android.com/devices/sensors/batching.html) for more details.) The events in the FIFO are delivered to the applications when the SoC wakes up. If the FIFO is too small to store all events, the older events are lost; the oldest data is dropped to accommodate the latest data. In the extreme case where the FIFO is nonexistent, all events generated while the SoC is in suspend mode are lost. One exception is the latest event from each on-change sensor: the last event [must be saved](https://source.android.com/devices/sensors/batching.html#precautions_to_take_when_batching_non-wake-up_on-change_sensors)outside of the FIFO so it cannot be lost.

As soon as the SoC gets out of suspend mode, all events from the FIFO are reported and operations resume as normal.

Applications using non-wake-up sensors should either hold a wake lock to ensure the system doesn't go to suspend, unregister from the sensors when they do not need them, or expect to lose events while the SoC is in suspend mode.

## Wake-up sensors

In opposition to non-wake-up sensors, wake-up sensors ensure that their data is delivered independently of the state of the SoC. While the SoC is awake, the wake-up sensors behave like non-wake-up-sensors. When the SoC is asleep, wake-up sensors must wake up the SoC to deliver events. They must still let the SoC go into suspend mode, but must also wake it up when an event needs to be reported. That is, the sensor must wake the SoC up and deliver the events before the maximum reporting latency has elapsed or the hardware FIFO gets full. See [Batching](https://source.android.com/devices/sensors/batching.html) for more details.

To ensure the applications have the time to receive the event before the SoC goes back to sleep, the driver must hold a "timeout wake lock" for 200 milliseconds each time an event is being reported. That is, the SoC should not be allowed to go back to sleep in the 200 milliseconds following a wake-up interrupt. This requirement will disappear in a future Android release, and we need this timeout wake lock until then.

## How to define wake-up and non-wake-up sensors?

Up to KitKat, whether a sensor was a wake-up or a non-wake-up sensor was dictated by the sensor type: most were non-wake-up sensors, with the exception of the [proximity](https://source.android.com/devices/sensors/sensor-types.html#proximity) sensor and the [significant motion detector](https://source.android.com/devices/sensors/sensor-types.html#significant_motion).

Starting in L, whether a given sensor is a wake-up sensor or not is specified by a flag in the sensor definition. Most sensors can be defined by pairs of wake-up and non-wake-up variants of the same sensor, in which case they must behave as two independent sensors, not interacting with one another. See [Interaction](https://source.android.com/devices/sensors/interaction.html) for more details.

Unless specified otherwise in the sensor type definition, it is recommended to implement one wake-up sensor and one non-wake-up sensor for each sensor type listed in [Sensor types](https://source.android.com/devices/sensors/sensor-types.html). In each sensor type definition, see what sensor (wake-up or non-wake-up) will be returned by SensorManager.getDefaultSensor(sensorType). It is the sensor that most applications will use.

# Power consumption

## IN THIS DOCUMENT

1. [Low-power sensors](https://source.android.com/devices/sensors/power-use.html#low_power_sensors)
2. [Power measurement process](https://source.android.com/devices/sensors/power-use.html#power_measurement_process)

## Low-power sensors

Some sensor types are defined as being low power. Low-power sensors must function at low power, with their processing done in the hardware. This means they should not require the SoC to be running. Here are some low-power sensor types:

* Geomagnetic rotation vector
* Significant motion
* Step counter
* Step detector
* Tilt detector

They are accompanied by a low-power () icon in the [Composite sensor type summary](https://source.android.com/devices/sensors/sensor-types.html#composite_sensor_type_summary) table.

* These sensor types cannot be implemented at high power as their primary benefit is low battery use. These sensors are expected to be activated for very long periods, possibly 24/7. It is better to not implement a low-power sensor at all rather than implement it as high power, as it would cause dramatic battery drain.
* Composite low-power sensor types, such as the step detector, must have their processing conducted in the hardware.
* See the CDD for specific power requirements, and expect tests in CTS to verify those power requirements.

## Power measurement process

* The power is measured at the battery. For values in milliWatts, we use the nominal voltage of the battery, meaning a 1mA current at 4V must be counted as 4mW.
* The power is measured when the SoC is asleep, and averaged over a few seconds of the SoC being asleep, so that periodic spikes in power from the sensor chips are taken into account.
* For one-shot wake-up sensors, the power is measured while the sensor doesn’t trigger (so it doesn’t wake the SoC up). Similarly, for other sensors, the power is measured while the sensor data is stored in the hardware FIFO, so the SoC is not woken up.
* The power normally is measured as a delta with when no sensor is activated. When several sensors are activated, the delta in power must be no greater than the sum of the power of each activated sensor. If an accelerometer consumes 0.5mA and a step detector consumes 0.5mA, then activating both at the same time must consume less than 0.5+0.5=1mA.

# Interaction

From the perspective of Android applications, every Android sensor is an independent entity, meaning there is no interaction between the different sensors.

* This is true even though several Android sensors might share the same underlying physical sensor
* For example: step counter, significant motion and accelerometer, all relying on the same physical accelerometer, must be able to work concurrently
* This is also true for wake-up and non-wake-up versions of the same sensor

Android sensors must be able to work simultaneously and independently of one another. That is, any action on one Android sensor must not impact the behavior of the other sensors.

Specifically, at the HAL level:

* activating a sensor
* deactivating a sensor
* changing the sampling frequency of a sensor
* changing the maximum reporting latency of a sensor

cannot cause:

* another activated sensor to stop working
* another activated sensor to change sampling rate
* another activated sensor to decrease the quality of its measurements
* another non-activated sensor to start delivering events

Nor can any of the actions above prevent actions (activation, deactivation, and parameter changes) on another sensor from succeeding. For instance, whether we can activate the step counter must be independent of whether the accelerometer is currently activated.

As another important example, a wake-up sensor activated at 5Hz must generate events at around 5Hz, even if its non-wake-up variant is being activated at 100Hz.

# HAL interface

The HAL interface, declared in [sensors.h](https://source.android.com/devices/halref/sensors_8h.html), represents the interface between the Android [framework](https://source.android.com/devices/sensors/sensor-stack.html#framework) and the hardware-specific software. A HAL implementation must define each function declared in sensors.h. The main functions are:

* get\_sensors\_list - Returns the list of all sensors.
* activate - Starts or stops a sensor.
* batch - Sets a sensor’s parameters such as sampling frequency and maximum reporting latency.
* setDelay - Used only in HAL version 1.0. Sets the sampling frequency for a given sensor.
* flush - Flushes the FIFO of the specified sensor and reports a flush complete event when this is done.
* poll - Returns available sensor events.

The implementation must be thread safe and allow these functions to be called from different threads.

The interface also defines several types used by those functions. The main types are:

* sensors\_module\_t
* sensors\_poll\_device\_t
* sensor\_t
* sensors\_event\_t

In addition to the sections below, see [sensors.h](https://source.android.com/devices/halref/sensors_8h.html) for more information on those types.

## get\_sensors\_list(list)

int (\*get\_sensors\_list)(struct sensors\_module\_t\* module, struct sensor\_t  
  const\*\* list);

Provides the list of sensors implemented by the HAL. See [sensor\_t](https://source.android.com/devices/sensors/hal-interface.html#sensor_t) for details on how the sensors are defined.

The order in which the sensors appear in the list is the order in which the sensors will be reported to the applications. Usually, the base sensors appear first, followed by the composite sensors.

If several sensors share the same sensor type and wake-up property, the first one in the list is called the “default” sensor. It is the one returned by getDefaultSensor(int sensorType, bool wakeUp).

This function returns the number of sensors in the list.

## activate(sensor, true/false)

int (\*activate)(struct sensors\_poll\_device\_t \*dev, int sensor\_handle, int  
  enabled);

Activates or deactivates a sensor.

sensor\_handle is the handle of the sensor to activate/deactivate. A sensor’s handle is defined by the handle field of its [sensor\_t](https://source.android.com/devices/sensors/hal-interface.html#sensor_t) structure.

enabled is set to 1 to enable or 0 to disable the sensor.

One-shot sensors deactivate themselves automatically upon receiving an event, and they must still accept to be deactivated through a call to activate(..., enabled=0).

Non-wake-up sensors never prevent the SoC from going into suspend mode; that is, the HAL shall not hold a partial wake-lock on behalf of applications.

Wake-up sensors, when delivering events continuously, can prevent the SoC from going into suspend mode, but if no event needs to be delivered, the partial wake-lock must be released.

If enabled is 1 and the sensor is already activated, this function is a no-op and succeeds.

If enabled is 0 and the sensor is already deactivated, this function is a no-op and succeeds.

This function returns 0 on success and a negative error number otherwise.

## batch(sensor, flags, sampling period, maximum report latency)

int (\*batch)(  
     struct sensors\_poll\_device\_1\* dev,  
     int sensor\_handle,  
     int flags,  
     int64\_t sampling\_period\_ns,  
     int64\_t max\_report\_latency\_ns);

Sets a sensor’s parameters, including [sampling frequency](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns) and [maximum report latency](https://source.android.com/devices/sensors/hal-interface.html#max_report_latency_ns). This function can be called while the sensor is activated, in which case it must not cause any sensor measurements to be lost: Transitioning from one sampling rate to the other cannot cause lost events, nor can transitioning from a high maximum report latency to a low maximum report latency.

sensor\_handle is the handle of the sensor to configure.

flags is currently unused.

sampling\_period\_ns is the sampling period at which the sensor should run, in nanoseconds. See [sampling\_period\_ns](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns) for more details.

max\_report\_latency\_ns is the maximum time by which events can be delayed before being reported through the HAL, in nanoseconds. See the [max\_report\_latency\_ns](https://source.android.com/devices/sensors/hal-interface.html#max_report_latency_ns) paragraph for more details.

This function returns 0 on success and a negative error number otherwise.

### sampling\_period\_ns

What the sampling\_period\_ns parameter means depends on the specified sensor's reporting mode:

* Continuous: sampling\_period\_ns is the sampling rate, meaning the rate at which events are generated.
* On-change: sampling\_period\_ns limits the sampling rate of events, meaning events are generated no faster than every sampling\_period\_ns nanoseconds. There might be periods longer than sampling\_period\_ns where no event is generated if the measured values do not change for long periods. See [on-change](https://source.android.com/devices/sensors/report-modes.html#on-change) reporting mode for more details.
* One-shot: sampling\_period\_ns is ignored. It has no effect.
* Special: See the specific [sensor type descriptions](https://source.android.com/devices/sensors/sensor-types.html) for details on how sampling\_period\_ns is used for special sensors.

See [Reporting modes](https://source.android.com/devices/sensors/report-modes.html) for more information about the impact of sampling\_period\_ns in the different modes.

For continuous and on-change sensors,

* if sampling\_period\_ns is less than sensor\_t.minDelay, then the HAL implementation must silently clamp it to max(sensor\_t.minDelay, 1ms). Android does not support the generation of events at more than 1000Hz.
* if sampling\_period\_ns is greater than sensor\_t.maxDelay, then the HAL implementation must silently truncate it to sensor\_t.maxDelay.

Physical sensors sometimes have limitations on the rates at which they can run and the accuracy of their clocks. To account for this, we allow the actual sampling frequency to differ from the requested frequency, as long as it satisfies the requirements in the table below.

|  |  |
| --- | --- |
| If the requested frequency is | Then the actual frequency must be |
| below min frequency (<1/maxDelay) | between 90% and 110% of the min frequency |
| between min and max frequency | between 90% and 220% of the requested frequency |
| above max frequency (>1/minDelay) | between 90% and 110% of the max frequency  and below 1100Hz |

Note that this contract is valid only at the HAL level, where there is always a single client. At the SDK level, applications might get different rates, due to the multiplexing happening in the Framework. See [Framework](https://source.android.com/devices/sensors/sensor-stack.html#framework) for more details.

### max\_report\_latency\_ns

max\_report\_latency\_ns sets the maximum time in nanoseconds, by which events can be delayed and stored in the hardware FIFO before being reported through the HAL while the SoC is awake.

A value of zero signifies that the events must be reported as soon as they are measured, either skipping the FIFO altogether, or emptying the FIFO as soon as one event from this sensor is present in it.

For example, an accelerometer activated at 50Hz with max\_report\_latency\_ns=0 will trigger interrupts 50 times per second when the SoC is awake.

When max\_report\_latency\_ns>0, sensor events do not need to be reported as soon as they are detected. They can be temporarily stored in the hardware FIFO and reported in batches, as long as no event is delayed by more than max\_report\_latency\_ns nanoseconds. That is, all events since the previous batch are recorded and returned at once. This reduces the amount of interrupts sent to the SoC and allows the SoC to switch to a lower power mode (idle) while the sensor is capturing and batching data.

Each event has a timestamp associated with it. Delaying the time at which an event is reported does not impact the event timestamp. The timestamp must be accurate and correspond to the time at which the event physically happened, not the time it is being reported.

Allowing sensor events to be stored temporarily in the hardware FIFO does not modify the behavior of poll: events from different sensors can be interleaved, and as usual, all events from the same sensor are time-ordered.

See [Batching](https://source.android.com/devices/sensors/batching.html) for more details on sensor batching, including behaviors in suspend mode and out of suspend mode.

## setDelay(sensor, sampling period)

int (\*setDelay)(  
     struct sensors\_poll\_device\_t \*dev,  
     int sensor\_handle,  
     int64\_t sampling\_period\_ns);

After HAL version 1.0, this function is deprecated and is never called. Instead, the batch function is called to set the sampling\_period\_ns parameter.

In HAL version 1.0, setDelay was used instead of batch to set [sampling\_period\_ns](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns).

## flush(sensor)

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

Add a [flush complete event](https://source.android.com/devices/sensors/hal-interface.html#metadata_flush_complete_events) to the end of the hardware FIFO for the specified sensor and flushes the FIFO; those events are delivered as usual (i.e.: as if the maximum reporting latency had expired) and removed from the FIFO.

The flush happens asynchronously (i.e.: this function must return immediately). If the implementation uses a single FIFO for several sensors, that FIFO is flushed and the flush complete event is added only for the specified sensor.

If the specified sensor has no FIFO (no buffering possible), or if the FIFO, was empty at the time of the call, flushmust still succeed and send a flush complete event for that sensor. This applies to all sensors other than one-shot sensors.

When flush is called, even if a flush event is already in the FIFO for that sensor, an additional one must be created and added to the end of the FIFO, and the FIFO must be flushed. The number of flush calls must be equal to the number of flush complete events created.

flush does not apply to [one-shot](https://source.android.com/devices/sensors/report-modes.html#one-shot) sensors; if sensor\_handle refers to a one-shot sensor, flush must return -EINVALand not generate any flush complete metadata event.

This function returns 0 on success, -EINVAL if the specified sensor is a one-shot sensor or wasn’t enabled, and a negative error number otherwise.

## poll()

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

Returns an array of sensor data by filling the data argument. This function must block until events are available. It will return the number of events read on success, or a negative error number in case of an error.

The number of events returned in data must be less or equal to the count argument. This function shall never return 0 (no event).

## Sequence of calls

When the device boots, get\_sensors\_list is called.

When a sensor gets activated, the batch function will be called with the requested parameters, followed by activate(..., enable=1).

Note that in HAL version 1\_0, the order was the opposite: activate was called first, followed by set\_delay.

When the requested characteristics of a sensor are changing while it is activated, the batch function is called.

flush can be called at any time, even on non-activated sensors (in which case it must return -EINVAL)

When a sensor gets deactivated, activate(..., enable=0) will be called.

In parallel to those calls, the poll function will be called repeatedly to request data. poll can be called even when no sensors are activated.

## sensors\_module\_t

sensors\_module\_t is the type used to create the Android hardware module for the sensors. The implementation of the HAL must define an object HAL\_MODULE\_INFO\_SYM of this type to expose the [get\_sensors\_list](https://source.android.com/devices/sensors/hal-interface.html#get_sensors_list_list) function. See the definition of sensors\_module\_t in [sensors.h](https://source.android.com/devices/halref/sensors_8h.html) and the definition of hw\_module\_t for more information.

## sensors\_poll\_device\_t / sensors\_poll\_device\_1\_t

sensors\_poll\_device\_1\_t contains the rest of the methods defined above: activate, batch, flush and poll. Its common field (of type [hw\_device\_t](https://source.android.com/devices/halref/structhw__device__t.html)) defines the version number of the HAL.

## sensor\_t

sensor\_t represents an [Android sensor](https://source.android.com/devices/sensors/index.html). Here are some of its important fields:

**name:** A user-visible string that represents the sensor. This string often contains the part name of the underlying sensor, the type of the sensor, and whether it is a wake-up sensor. For example, “LIS2HH12 Accelerometer”, “MAX21000 Uncalibrated Gyroscope”, “BMP280 Wake-up Barometer”, “MPU6515 Game Rotation Vector”

**handle:** The integer used to refer to the sensor when registering to it or generating events from it.

**type:** The type of the sensor. See the explanation of sensor type in [What are Android sensors?](https://source.android.com/devices/sensors/index.html) for more details, and see [Sensor types](https://source.android.com/devices/sensors/sensor-types.html) for official sensor types. For non-official sensor types, type must start with SENSOR\_TYPE\_DEVICE\_PRIVATE\_BASE

**stringType:** The type of the sensor as a string. When the sensor has an official type, set to SENSOR\_STRING\_TYPE\_\*. When the sensor has a manufacturer specific type, stringType must start with the manufacturer reverse domain name. For example, a sensor (say a unicorn detector) defined by the Cool-product team at Fictional-Company could use stringType=”com.fictional\_company.cool\_product.unicorn\_detector”. The stringType is used to uniquely identify non-official sensors types. See [sensors.h](https://source.android.com/devices/halref/sensors_8h.html) for more information on types and string types.

**requiredPermission:** A string representing the permission that applications must possess to see the sensor, register to it and receive its data. An empty string means applications do not require any permission to access this sensor. Some sensor types like the [heart rate monitor](https://source.android.com/devices/sensors/sensor-types.html#heart_rate) have a mandatory requiredPermission. All sensors providing sensitive user information (such as the heart rate) must be protected by a permission.

**flags:** Flags for this sensor, defining the sensor’s reporting mode and whether the sensor is a wake-up sensor or not. For example, a one-shot wake-up sensor will have flags = SENSOR\_FLAG\_ONE\_SHOT\_MODE | SENSOR\_FLAG\_WAKE\_UP. The bits of the flag that are not used in the current HAL version must be left equal to 0.

**maxRange:** The maximum value the sensor can report, in the same unit as the reported values. The sensor must be able to report values without saturating within [-maxRange; maxRange]. Note that this means the total range of the sensor in the generic sense is 2\*maxRange. When the sensor reports values over several axes, the range applies to each axis. For example, a “+/- 2g” accelerometer will report maxRange = 2\*9.81 = 2g.

**resolution:** The smallest difference in value that the sensor can measure. Usually computed based on maxRangeand the number of bits in the measurement.

**power:** The power cost of enabling the sensor, in milliAmps. This is nearly always more that the power consumption reported in the datasheet of the underlying sensor. See [Base sensors != physical sensors](https://source.android.com/devices/sensors/sensor-types.html#base_sensors_=_not_equal_to_physical_sensors) for more details and see [Power measurement process](https://source.android.com/devices/sensors/power-use.html#power_measurement_process) for details on how to measure the power consumption of a sensor. If the sensor’s power consumption depends on whether the device is moving, the power consumption while moving is the one reported in the power field.

**minDelay:** For continuous sensors, the sampling period, in microseconds, corresponding to the fastest rate the sensor supports. See [sampling\_period\_ns](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns) for details on how this value is used. Beware that minDelay is expressed in microseconds while sampling\_period\_ns is in nanoseconds. For on-change and special reporting mode sensors, unless otherwise specified, minDelay must be 0. For one-shot sensors, it must be -1.

**maxDelay:** For continuous and on-change sensors, the sampling period, in microseconds, corresponding to the slowest rate the sensor supports. See [sampling\_period\_ns](https://source.android.com/devices/sensors/hal-interface.html#sampling_period_ns) for details on how this value is used. Beware that maxDelay is expressed in microseconds while sampling\_period\_ns is in nanoseconds. For special and one-shot sensors, maxDelay must be 0.

**fifoReservedEventCount:** The number of events reserved for this sensor in the hardware FIFO. If there is a dedicated FIFO for this sensor, then fifoReservedEventCount is the size of this dedicated FIFO. If the FIFO is shared with other sensors, fifoReservedEventCount is the size of the part of the FIFO that is reserved for that sensor. On most shared-FIFO systems, and on systems that do not have a hardware FIFO this value is 0.

**fifoMaxEventCount:** The maximum number of events that could be stored in the FIFOs for this sensor. This is always greater or equal to fifoReservedEventCount. This value is used to estimate how quickly the FIFO will get full when registering to the sensor at a specific rate, supposing no other sensors are activated. On systems that do not have a hardware FIFO, fifoMaxEventCount is 0. See [Batching](https://source.android.com/devices/sensors/batching.html) for more details.

For sensors with an official sensor type, some of the fields are overwritten by the framework. For example, [accelerometer](https://source.android.com/devices/sensors/sensor-types.html#accelerometer) sensors are forced to have a continuous reporting mode, and [heart rate](https://source.android.com/devices/sensors/sensor-types.html#heart_rate) monitors are forced to be protected by the SENSOR\_PERMISSION\_BODY\_SENSORS permission.

## sensors\_event\_t

Sensor events generated by Android sensors and reported through the [poll](https://source.android.com/devices/sensors/hal-interface.html#poll) function are of type sensors\_event\_t. Here are some important fields of sensors\_event\_t:

**version:** Must be sizeof(struct sensors\_event\_t)

**sensor:** The handle of the sensor that generated the event, as defined by sensor\_t.handle.

**type:** The sensor type of the sensor that generated the event, as defined by sensor\_t.type.

**timestamp:** The timestamp of the event in nanoseconds. This is the time the event happened (a step was taken, or an accelerometer measurement was made), not the time the event was reported. timestamp must be synchronized with the elapsedRealtimeNano clock, and in the case of continuous sensors, the jitter must be small. Timestamp filtering is sometimes necessary to satisfy the CDD requirements, as using only the SoC interrupt time to set the timestamps causes too high jitter, and using only the sensor chip time to set the timestamps can cause de-synchronization from the elapsedRealtimeNano clock, as the sensor clock drifts.

**data and overlapping fields:** The values measured by the sensor. The meaning and units of those fields are specific to each sensor type. See [sensors.h](https://source.android.com/devices/halref/sensors_8h.html) and the definition of the different [Sensor types](https://source.android.com/devices/sensors/sensor-types.html) for a description of the data fields. For some sensors, the accuracy of the readings is also reported as part of the data, through a status field. This field is only piped through for those select sensor types, appearing at the SDK layer as an accuracy value. For those sensors, the fact that the status field must be set is mentioned in their [sensor type](https://source.android.com/devices/sensors/sensor-types.html)definition.

### Metadata flush complete events

Metadata events have the same type as normal sensor events: sensors\_event\_meta\_data\_t = sensors\_event\_t. They are returned together with other sensor events through poll. They possess the following fields:

**version:** Must be META\_DATA\_VERSION

**type:** Must be SENSOR\_TYPE\_META\_DATA

**sensor, reserved, and timestamp**: Must be 0

**meta\_data.what:** Contains the metadata type for this event. There is currently a single valid metadata type: META\_DATA\_FLUSH\_COMPLETE.

META\_DATA\_FLUSH\_COMPLETE events represent the completion of the flush of a sensor FIFO. When meta\_data.what=META\_DATA\_FLUSH\_COMPLETE, meta\_data.sensor must be set to the handle of the sensor that has been flushed. They are generated when and only when flush is called on a sensor. See the section on the [flush](https://source.android.com/devices/sensors/hal-interface.html#flush_sensor)function for more information.

# Batching

## What is batching?

“Batching” refers to storing sensor events in a hardware FIFO before reporting them through the [HAL](https://source.android.com/devices/sensors/hal-interface.html) instead of reporting them immediately.

Batching can enable significant power savings by preventing the SoC from waking up to receive each event. Instead, the events can be grouped and processed together.

The bigger the FIFOs, the more power can be saved. Implementing batching is an exercise of trading off hardware memory for reduced power consumption.

Batching happens when a sensor possesses a hardware FIFO (sensor\_t.fifoMaxEventCount > 0) and we are in one of two situations:

* max\_report\_latency > 0, meaning the sensor events for this specific sensor can be delayed up to max\_report\_latency before being reported through the HAL.
* or the SoC is in suspend mode and the sensor is a non-wake-up sensor, meaning events must be stored while waiting for the SoC to wake up.

See the paragraph on the [HAL batch function](https://source.android.com/devices/sensors/hal-interface.html#batch_sensor_flags_sampling_period_maximum_report_latency) for more details.

The opposite of batching is the continuous operation, where events are not buffered, meaning they are reported immediately. Continuous operation corresponds to:

* when max\_report\_latency = 0 and the events can be delivered to the application, meaning
  + the SoC is awake
  + or the sensor is a wake-up sensor
* or when the sensor doesn’t have a hardware FIFO (sensor\_t.fifoMaxEventCount = 0), in which case
  + the events are reported if the SoC is awake or the sensor is a wake-up sensor
  + the events are lost when the SoC is asleep and the sensor is not a wake-up sensor

## Wake-up FIFOs and non-wake-up FIFOs

Sensor events from [wake-up sensors](https://source.android.com/devices/sensors/suspend-mode.html#wake-up_sensors) must be stored into a wake-up FIFO. There can be one wake-up FIFO per sensor, or, more commonly, one big shared wake-up FIFO where events from all wake-up sensors are interleaved. Other options are also possible, with for example some wake-up sensors having a dedicated FIFO, and the rest of the wake-up sensors all sharing the same one.

Similarly, sensor events from [non-wake-up sensors](https://source.android.com/devices/sensors/suspend-mode.html#non-wake-up_sensors) must be stored into a non-wake-up FIFOs, and there can be one or several non-wake-up FIFOs.

In all cases, wake-up sensor events and non-wake-up sensor events cannot be interleaved into the same FIFO. Wake-up events go in wake-up FIFOs, and non-wake-up events go in non-wake-up FIFOs.

For the wake-up FIFO, the “one big shared FIFO” design provides the best power benefits. For the non-wake-up FIFO, there is no preference between the “one big shared FIFO” and “several small reserved FIFOs”. See [FIFO allocation priority](https://source.android.com/devices/sensors/batching.html#fifo_allocation_priority) for suggestions on how to dimension each FIFO.

## Behavior outside of suspend mode

When the SoC is awake (not in suspend mode), the events can be stored temporarily in their FIFO, as long as they are not delayed by more than max\_report\_latency.

As long as the SoC doesn’t enter the suspend mode, no event shall be dropped or lost. If internal hardware FIFOs is getting full before max\_report\_latency elapsed, then events are reported at that point to ensure that no event is lost.

If several sensors share the same FIFO and the max\_report\_latency of one of them elapses, all events from the FIFO are reported, even if the max\_report\_latency of the other sensors didn’t elapse yet. The general goal is to reduce the number of times batches of events must be reported, so as soon as one event must be reported, all events from all sensors can be reported.

For example, if the following sensors are activated:

* accelerometer batched with max\_report\_latency = 20s
* gyroscope batched with max\_report\_latency = 5s

Then the accelerometer batches can be reported at the same time the gyroscope batches are reported (every 5 seconds), even if the accelerometer and the gyroscope do not share the same FIFO.

## Behavior in suspend mode

Batching is particularly beneficial when wanting to collect sensor data in the background without keeping the SoC awake. Because the sensor drivers and HAL implementation are not allowed to hold a wake-lock\*, the SoC can enter the suspend mode even while sensor data is being collected.

The behavior of sensors while the SoC is suspended depends on whether the sensor is a wake-up sensor. See [Wake-up sensors](https://source.android.com/devices/sensors/suspend-mode.html#wake-up_sensors) for some details.

When a non-wake-up FIFO fills up, it must wrap around and behave like a circular buffer, overwriting older events: the new events replace the old ones. max\_report\_latency has no impact on non-wake-up FIFOs while in suspend mode.

When a wake-up FIFO fills up, or when the max\_report\_latency of one of the wake-up sensor elapsed, the hardware must wake up the SoC and report the data.

In both cases (wake-up and non-wake-up), as soon as the SoC comes out of suspend mode, a batch is produced with the content of all FIFOs, even if max\_report\_latency of some sensors didn’t elapse yet. This minimizes the risk of having to wake-up the SoC again soon if it goes back to suspend. Hence, it minimizes power consumption.

\*One notable exception of drivers not being allowed to hold a wake lock is when a wake-up sensor with [continuous reporting mode](https://source.android.com/devices/sensors/report-modes.html#continuous) is activated with max\_report\_latency < 1 second. In that case, the driver can hold a wake lock because the SoC would anyway not have the time to enter the suspend mode, as it would be awoken by a wake-up event before reaching the suspend mode.

## Precautions to take when batching wake-up sensors

Depending on the device, it might take a few milliseconds for the SoC to entirely come out of suspend and start flushing the FIFO. Enough head room must be allocated in the FIFO to allow the device to entirely come out of suspend without the wake-up FIFO overflowing. No events shall be lost, and the max\_report\_latency must be respected.

## Precautions to take when batching non-wake-up on-change sensors

On-change sensors only generate events when the value they are measuring is changing. If the measured value changes while the SoC is in suspend mode, applications expect to receive an event as soon as the SoC wakes up. Because of this, batching of [non-wake-up](https://source.android.com/devices/sensors/suspend-mode.html#non-wake-up_sensors) on-change sensor events must be performed carefully if the sensor shares its FIFO with other sensors. The last event generated by each on-change sensor must always be saved outside of the shared FIFO so it can never be overwritten by other events. When the SoC wakes up, after all events from the FIFO have been reported, the last on-change sensor event must be reported.

Here is a situation we want to avoid:

1. An application registers to the non-wake-up step counter (on-change) and the non-wake-up accelerometer (continuous), both sharing the same FIFO
2. The application receives a step counter event “step\_count=1000 steps”
3. The SoC goes to suspend
4. The user walks 20 steps, causing step counter and accelerometer events to be interleaved, the last step counter event being “step\_count = 1020 steps”
5. The user doesn’t move for a long time, causing accelerometer events to continue accumulating in the FIFO, eventually overwriting every step\_count event in the shared FIFO
6. SoC wakes up and all events from the FIFO are sent to the application
7. The application receives only accelerometer events and thinks that the user didn’t walk (bad!)

By saving the last step counter event outside of the FIFO, the HAL can report this event when the SoC wakes up, even if all other step counter events were overwritten by accelerometer events. This way, the application receives “step\_count = 1020 steps” when the SoC wakes up.

## Implementing batching

Batching cannot be emulated in software. It must be implemented entirely in hardware, with hardware FIFOs. In particular, it cannot be implemented on the SoC, for example in the HAL implementation, as this would be counter-productive. The goal here is to save significant amounts of power. Batching must be implemented without the aid of the SoC, which should be allowed to be in suspend mode during batching.

max\_report\_latency can be modified at any time, in particular while the specified sensor is already enabled; and this shall not result in the loss of events.

## FIFO allocation priority

On platforms in which hardware FIFO size is limited, the system designers may have to choose how much FIFO to reserve for each sensor. To help with this choice, here is a list of applications made possible when batching is implemented on the different sensors.

### High value: Low power pedestrian dead reckoning

Target batching time: 1 to 10 minutes

Sensors to batch:

* Wake-up Step detector
* Wake-up Game rotation vector at 5Hz
* Wake-up Barometer at 5Hz
* Wake-up Uncalibrated Magnetometer at 5Hz

Batching this data allows performing pedestrian dead reckoning while letting the SoC go to suspend.

### High value: Medium power intermittent activity/gesture recognition

Target batching time: 3 seconds

Sensors to batch: Non-wake-up Accelerometer at 50Hz

Batching this data allows periodically recognizing arbitrary activities and gestures without having to keep the SoC awake while the data is collected.

### Medium value: Medium power continuous activity/gesture recognition

Target batching time: 1 to 3 minutes

Sensors to batch: Wake-up Accelerometer at 50Hz

Batching this data allows continuously recognizing arbitrary activities and gestures without having to keep the SoC awake while the data is collected.

### Medium-high value: Interrupt load reduction

Target batching time: < 1 second

Sensors to batch: any high frequency sensor, usually non-wake-up.

If the gyroscope is set at 240Hz, even batching just 10 gyro events can reduce the number of interrupts from 240/second to 24/second.

### Medium value: Continuous low frequency data collection

Target batching time: 1 to 10 minutes

Sensors to batch:

* Wake-up barometer at 1Hz,
* Wake-up humidity sensor at 1Hz
* Other low frequency wake-up sensors at similar rates

Allows creating monitoring applications at low power.

### Medium-low value: Continuous full-sensors collection

Target batching time: 1 to 10 minutes

Sensors to batch: all wake-up sensors, at high frequencies

Allows full collection of sensor data while leaving the SoC in suspend mode. Only to consider if FIFO space is not an issue.

## Base sensors

Base sensor types are named after the physical sensors they represent. These sensors relay data from a single physical sensor (as opposed to composite sensors that generate data out of other sensors). Examples of base sensor types include:

* SENSOR\_TYPE\_ACCELEROMETER
* SENSOR\_TYPE\_GYROSCOPE
* SENSOR\_TYPE\_MAGNETOMETER

**Note:** For details on each Android sensor type, review the following sections.

However, base sensors are not equal to and should not be confused with their underlying physical sensor. The data from a base sensor is **not** the raw output of the physical sensor because corrections (such as bias compensation and temperature compensation) are applied.

For example, the characteristics of a base sensor might be different from the characteristics of its underlying physical sensor in the following use cases:

* A gyroscope chip rated to have a bias range of 1 deg/sec.
  + After factory calibration, temperature compensation and bias compensation are applied, the actual bias of the Android sensor will be reduced, may be to a point where the bias is guaranteed to be below 0.01deg/sec.
  + In this situation, we say that the Android sensor has a bias below 0.01 deg/sec, even though the data sheet of the underlying sensor said 1 deg/sec.
* A barometer with a power consumption of 100uW.
  + Because the generated data needs to be transported from the chip to the SoC, the actual power cost to gather data from the barometer Android sensor might be much higher, for example 1000uW.
  + In this situation, we say that the Android sensor has a power consumption of 1000uW, even though the power consumption measured at the barometer chip leads is 100uW.
* A magnetometer that consumes 100uW when calibrated, but consumes more when calibrating.
  + Its calibration routine might require activating the gyroscope, consuming 5000uW, and running some algorithm, costing another 900uW.
  + In this situation, we say that the maximum power consumption of the (magnetometer) Android sensor is 6000uW.
  + In this case, the average power consumption is the more useful measure, and it is what is reported in the sensor static characteristics through the HAL.

### Accelerometer

Reporting-mode: [*Continuous*](https://source.android.com/devices/sensors/report-modes.html#continuous)

getDefaultSensor(SENSOR\_TYPE\_ACCELEROMETER) returns a non-wake-up sensor

An accelerometer sensor reports the acceleration of the device along the 3 sensor axes. The measured acceleration includes both the physical acceleration (change of velocity) and the gravity. The measurement is reported in the x, y and z fields of sensors\_event\_t.acceleration.

All values are in SI units (m/s^2) and measure the acceleration of the device minus the force of gravity along the 3 sensor axes.

Here are examples:

* The norm of (x, y, z) should be close to 0 when in free fall.
* When the device lies flat on a table and is pushed on its left side toward the right, the x acceleration value is positive.
* When the device lies flat on a table, the acceleration value along z is +9.81 alo, which corresponds to the acceleration of the device (0 m/s^2) minus the force of gravity (-9.81 m/s^2).
* When the device lies flat on a table and is pushed toward the sky, the acceleration value is greater than +9.81, which corresponds to the acceleration of the device (+A m/s^2) minus the force of gravity (-9.81 m/s^2).

The readings are calibrated using:

* temperature compensation
* online bias calibration
* online scale calibration

The bias and scale calibration must only be updated while the sensor is deactivated, so as to avoid causing jumps in values during streaming.

The accelerometer also reports how accurate it expects its readings to be through sensors\_event\_t.acceleration.status. See the [SensorManager](http://developer.android.com/reference/android/hardware/SensorManager.html)’s [SENSOR\_STATUS\_\*](http://developer.android.com/reference/android/hardware/SensorManager.html#SENSOR_STATUS_ACCURACY_HIGH)constants for more information on possible values for this field.

### Ambient temperature

Reporting-mode: [*On-change*](https://source.android.com/devices/sensors/report-modes.html#on-change)

getDefaultSensor(SENSOR\_TYPE\_AMBIENT\_TEMPERATURE) returns a non-wake-up sensor

This sensor provides the ambient (room) temperature in degrees Celsius.

### Magnetic field sensor

Reporting-mode: [*Continuous*](https://source.android.com/devices/sensors/report-modes.html#continuous)

getDefaultSensor(SENSOR\_TYPE\_MAGNETIC\_FIELD) returns a non-wake-up sensor

SENSOR\_TYPE\_GEOMAGNETIC\_FIELD == SENSOR\_TYPE\_MAGNETIC\_FIELD

A magnetic field sensor (also known as magnetometer) reports the ambient magnetic field, as measured along the 3 sensor axes.

The measurement is reported in the x, y and z fields of sensors\_event\_t.magnetic and all values are in micro-Tesla (uT).

The magnetometer also reports how accurate it expects its readings to be through sensors\_event\_t.magnetic.status. See the [SensorManager](http://developer.android.com/reference/android/hardware/SensorManager.html)’s [SENSOR\_STATUS\_\*](http://developer.android.com/reference/android/hardware/SensorManager.html#SENSOR_STATUS_ACCURACY_HIGH) constants for more information on possible values for this field.

The readings are calibrated using:

* temperature compensation
* factory (or online) soft-iron calibration
* online hard-iron calibration

### Gyroscope

Reporting-mode: [*Continuous*](https://source.android.com/devices/sensors/report-modes.html#continuous)

getDefaultSensor(SENSOR\_TYPE\_GYROSCOPE) returns a non-wake-up sensor

A gyroscope sensor reports the rate of rotation of the device around the 3 sensor axes.

Rotation is positive in the counterclockwise direction (right-hand rule). That is, an observer looking from some positive location on the x, y or z axis at a device positioned on the origin would report positive rotation if the device appeared to be rotating counter clockwise. Note that this is the standard mathematical definition of positive rotation and does not agree with the aerospace definition of roll.

The measurement is reported in the x, y and z fields of sensors\_event\_t.gyro and all values are in radians per second (rad/s).

The readings are calibrated using:

* temperature compensation
* factory (or online) scale compensation
* online bias calibration (to remove drift)

The gyroscope also reports how accurate it expects its readings to be through sensors\_event\_t.gyro.status. See the [SensorManager](http://developer.android.com/reference/android/hardware/SensorManager.html)’s [SENSOR\_STATUS\_\*](http://developer.android.com/reference/android/hardware/SensorManager.html#SENSOR_STATUS_ACCURACY_HIGH) constants for more information on possible values for this field.

The gyroscope cannot be emulated based on magnetometers and accelerometers, as this would cause it to have reduced local consistency and responsiveness. It must be based on a usual gyroscope chip.

### Heart Rate

Reporting-mode: [*On-change*](https://source.android.com/devices/sensors/report-modes.html#on-change)

getDefaultSensor(SENSOR\_TYPE\_HEART\_RATE) returns a non-wake-up sensor

A heart rate sensor reports the current heart rate of the person touching the device.

The current heart rate in beats per minute (BPM) is reported in sensors\_event\_t.heart\_rate.bpm and the status of the sensor is reported in sensors\_event\_t.heart\_rate.status. See the [SensorManager](http://developer.android.com/reference/android/hardware/SensorManager.html)’s [SENSOR\_STATUS\_\*](http://developer.android.com/reference/android/hardware/SensorManager.html#SENSOR_STATUS_ACCURACY_HIGH) constants for more information on possible values for this field. In particular, upon the first activation, unless the device is known to not be on the body, the status field of the first event must be set to SENSOR\_STATUS\_UNRELIABLE. Because this sensor is on-change, events are generated when and only when heart\_rate.bpm or heart\_rate.status have changed since the last event. The events are generated no faster than every sampling\_period.

sensor\_t.requiredPermission is always SENSOR\_PERMISSION\_BODY\_SENSORS.

### Light

Reporting-mode: [*On-change*](https://source.android.com/devices/sensors/report-modes.html#on-change)

getDefaultSensor(SENSOR\_TYPE\_LIGHT) returns a non-wake-up sensor

A light sensor reports the current illumination in SI lux units.

The measurement is reported in sensors\_event\_t.light.

### Proximity

Reporting-mode: [*On-change*](https://source.android.com/devices/sensors/report-modes.html#on-change)

Usually defined as a wake-up sensor

getDefaultSensor(SENSOR\_TYPE\_PROXIMITY) returns a wake-up sensor

A proximity sensor reports the distance from the sensor to the closest visible surface.

Up to Android KitKat, the proximity sensors were always wake-up sensors, waking up the SoC when detecting a change in proximity. After Android KitKat, we advise to implement the wake-up version of this sensor first, as it is the one that is used to turn the screen on and off while making phone calls.

The measurement is reported in centimeters in sensors\_event\_t.distance. Note that some proximity sensors only support a binary "near" or "far" measurement. In this case, the sensor report its sensor\_t.maxRange value in the "far" state and a value less than sensor\_t.maxRange in the "near" state.

### Pressure

Reporting-mode: [*Continuous*](https://source.android.com/devices/sensors/report-modes.html#continuous)

getDefaultSensor(SENSOR\_TYPE\_PRESSURE) returns a non-wake-up sensor

A pressure sensor (also known as barometer) reports the atmospheric pressure in hectopascal (hPa).

The readings are calibrated using

* temperature compensation
* factory bias calibration
* factory scale calibration

The barometer is often used to estimate elevation changes. To estimate absolute elevation, the sea-level pressure (changing depending on the weather) must be used as a reference.

### Relative humidity

Reporting-mode: [*On-change*](https://source.android.com/devices/sensors/report-modes.html#on-change)

getDefaultSensor(SENSOR\_TYPE\_RELATIVE\_HUMIDITY) returns a non-wake-up sensor

A relative humidity sensor measures relative ambient air humidity and returns a value in percent.

## Composite sensor types

A composite sensor generates data by processing and/or fusing data from one or several physical sensors. (Any sensor that is not a base sensor is called a composite sensor.) Examples of composite sensors include:

* [Step detector](https://source.android.com/devices/sensors/sensor-types.html#step_detector) and [Significant motion](https://source.android.com/devices/sensors/sensor-types.html#significant_motion), which are usually based on an accelerometer, but could be based on other sensors as well, if the power consumption and accuracy was acceptable.
* [Game rotation vector](https://source.android.com/devices/sensors/sensor-types.html#game_rotation_vector), based on an accelerometer and a gyroscope.
* [Uncalibrated gyroscope](https://source.android.com/devices/sensors/sensor-types.html#gyroscope_uncalibrated), which is similar to the gyroscope base sensor, but with the bias calibration being reported separately instead of being corrected in the measurement.

As with base sensors, the characteristics of the composite sensors come from the characteristics of their final data. For example, the power consumption of a game rotation vector is probably equal to the sum of the power consumptions of the accelerometer chip, the gyroscope chip, the chip processing the data, and the buses transporting the data. As another example, the drift of a game rotation vector depends as much on the quality of the calibration algorithm as on the physical sensor characteristics.

The following table lists available composite sensor types. Each composite sensor relies on data from one or several physical sensors. Avoid choosing other underlying physical sensors to approximate results as they provide a poor user experience.

**Note:** When there is no gyroscope on the device (and only when there is no gyroscope), you may implement the rotation vector, linear acceleration, and gravity sensors without using the gyroscope.

Sensor type Category Underlying physical sensors Reporting mode

Game rotation vector

Attitude

Accelerometer, Gyroscope MUST NOT USE Magnetometer

Continuous

Geomagnetic rotation vector Low

power sensor

Attitude

Accelerometer, Magnetometer, MUST NOT USE Gyroscope

Continuous

Glance gesture Low power sensor

Interaction

Undefined

One-shot

Gravity

Attitude

Accelerometer, Gyroscope

Continuous

Gyroscope uncalibrated

Uncalibrated

Gyroscope

Continuous

Linear acceleration

Activity

Accelerometer, Gyroscope (if present) or Magnetometer (if gyro not present)

Continuous

Magnetic field uncalibrated

Uncalibrated

Magnetometer

Continuous

Orientation (deprecated)

Attitude

Accelerometer, Magnetometer PREFERRED Gyroscope

Continuous

Pick up gesture Low power sensor

Interaction

Undefined

One-shot

Rotation vector

Attitude

Accelerometer, Magnetometer, AND (when present) Gyroscope

Continuous

Significant motion Low power sensor

Activity

Accelerometer (or another as long as very low power)

One-shot

Step counter Low power sensor

Activity

Accelerometer

On-change

Step detector Low power sensor

Activity

Accelerometer

Special

Tilt detector Low power sensor

Activity

Accelerometer

Special

Wake up gesture Low power sensor

Interaction

Undefined

One-shot

Low power sensor = Low power sensor

Activity composite sensors

Linear acceleration

Underlying physical sensors: Accelerometer and (if present) Gyroscope (or magnetometer if gyroscope not present)

Reporting-mode: Continuous

getDefaultSensor(SENSOR\_TYPE\_LINEAR\_ACCELERATION) returns a non-wake-up sensor

A linear acceleration sensor reports the linear acceleration of the device in the sensor frame, not including gravity.

The output is conceptually: output of the accelerometer minus the output of the gravity sensor. It is reported in m/s^2 in the x, y and z fields of sensors\_event\_t.acceleration.

Readings on all axes should be close to 0 when the device is immobile.

If the device possesses a gyroscope, the linear acceleration sensor must use the gyroscope and accelerometer as input.

If the device doesn’t possess a gyroscope, the linear acceleration sensor must use the accelerometer and the magnetometer as input.

Significant motion

Underlying physical sensor: Accelerometer (or another as long as low power)

Reporting-mode: One-shot

Low-power

Implement only the wake-up version of this sensor.

getDefaultSensor(SENSOR\_TYPE\_SIGNIFICANT\_MOTION) returns a wake-up sensor

A significant motion detector triggers when the detecting a “significant motion”: a motion that might lead to a change in the user location.

Examples of such significant motions are:

walking or biking

sitting in a moving car, coach or train

Examples of situations that do not trigger significant motion:

phone in pocket and person is not moving

phone is on a table and the table shakes a bit due to nearby traffic or washing machine

At the high level, the significant motion detector is used to reduce the power consumption of location determination. When the localization algorithms detect that the device is static, they can switch to a low power mode, where they rely on significant motion to wake the device up when the user is changing location.

This sensor must be low power. It makes a tradeoff for power consumption that may result in a small amount of false negatives. This is done for a few reasons:

The goal of this sensor is to save power.

Triggering an event when the user is not moving (false positive) is costly in terms of power, so it should be avoided.

Not triggering an event when the user is moving (false negative) is acceptable as long as it is not done repeatedly. If the user has been walking for 10 seconds, not triggering an event within those 10 seconds is not acceptable.

Each sensor event reports 1 in sensors\_event\_t.data[0]

Step detector

Underlying physical sensor: Accelerometer (+ possibly others as long as low power)

Reporting-mode: Special (one event per step taken)

Low-power

getDefaultSensor(SENSOR\_TYPE\_STEP\_DETECTOR) returns a non-wake-up sensor

A step detector generates an event each time a step is taken by the user.

The timestamp of the event sensors\_event\_t.timestamp corresponds to when the foot hit the ground, generating a high variation in acceleration.

Compared to the step counter, the step detector should have a lower latency (less than 2 seconds). Both the step detector and the step counter detect when the user is walking, running and walking up the stairs. They should not trigger when the user is biking, driving or in other vehicles.

This sensor must be low power. That is, if the step detection cannot be done in hardware, this sensor should not be defined. In particular, when the step detector is activated and the accelerometer is not, only steps should trigger interrupts (not every accelerometer reading).

sampling\_period\_ns has no impact on step detectors.

Each sensor event reports 1 in sensors\_event\_t.data[0]

Step counter

Underlying physical sensor: Accelerometer (+ possibly others as long as low power)

Reporting-mode: On-change

Low-power

getDefaultSensor(SENSOR\_TYPE\_STEP\_COUNTER) returns a non-wake-up sensor

A step counter reports the number of steps taken by the user since the last reboot while activated.

The measurement is reported as a uint64\_t in sensors\_event\_t.step\_counter and is reset to zero only on a system reboot.

The timestamp of the event is set to the time when the last step for that event was taken.

See the Step detector sensor type for the signification of the time of a step.

Compared to the step detector, the step counter can have a higher latency (up to 10 seconds). Thanks to this latency, this sensor has a high accuracy; the step count after a full day of measures should be within 10% of the actual step count. Both the step detector and the step counter detect when the user is walking, running and walking up the stairs. They should not trigger when the user is biking, driving or in other vehicles.

The hardware must ensure the internal step count never overflows. The minimum size of the hardware's internal counter shall be 16 bits. In case of imminent overflow (at most every ~2^16 steps), the SoC can be woken up so the driver can do the counter maintenance.

As stated in Interaction, while this sensor operates, it shall not disrupt any other sensors, in particular, the accelerometer, which might very well be in use.

If a particular device cannot support these modes of operation, then this sensor type must not be reported by the HAL. ie: it is not acceptable to "emulate" this sensor in the HAL.

This sensor must be low power. That is, if the step detection cannot be done in hardware, this sensor should not be defined. In particular, when the step counter is activated and the accelerometer is not, only steps should trigger interrupts (not accelerometer data).

Tilt detector

Underlying physical sensor: Accelerometer (+ possibly others as long as low power)

Reporting-mode: Special

Low-power

Implement only the wake-up version of this sensor.

getDefaultSensor(SENSOR\_TYPE\_TILT\_DETECTOR) returns a wake-up sensor

A tilt detector generates an event each time a tilt event is detected.

A tilt event is defined by the direction of the 2-seconds window average gravity changing by at least 35 degrees since the activation or the last event generated by the sensor. Here is the algorithm:

reference\_estimated\_gravity = average of accelerometer measurements over the first second after activation or the estimated gravity when the last tilt event was generated.

current\_estimated\_gravity = average of accelerometer measurements over the last 2 seconds.

trigger when angle(reference\_estimated\_gravity, current\_estimated\_gravity) > 35 degrees

Large accelerations without a change in phone orientation should not trigger a tilt event. For example, a sharp turn or strong acceleration while driving a car should not trigger a tilt event, even though the angle of the average acceleration might vary by more than 35 degrees. Typically, this sensor is implemented with the help of only an accelerometer. Other sensors can be used as well if they do not increase the power consumption significantly. This is a low power sensor that should allow the SoC to go into suspend mode. Do not emulate this sensor in the HAL. Each sensor event reports 1 in sensors\_event\_t.data[0].

Attitude composite sensors

Rotation vector

Underlying physical sensors: Accelerometer, Magnetometer, and Gyroscope

Reporting-mode: Continuous

getDefaultSensor(SENSOR\_TYPE\_ROTATION\_VECTOR) returns a non-wake-up sensor

A rotation vector sensor reports the orientation of the device relative to the East-North-Up coordinates frame. It is usually obtained by integration of accelerometer, gyroscope, and magnetometer readings. The East-North-Up coordinate system is defined as a direct orthonormal basis where:

X points east and is tangential to the ground.

Y points north and is tangential to the ground.

Z points towards the sky and is perpendicular to the ground.

The orientation of the phone is represented by the rotation necessary to align the East-North-Up coordinates with the phone's coordinates. That is, applying the rotation to the world frame (X,Y,Z) would align them with the phone coordinates (x,y,z).

The rotation can be seen as rotating the phone by an angle theta around an axis rot\_axis to go from the reference (East-North-Up aligned) device orientation to the current device orientation. The rotation is encoded as the four unit-less x, y, z, w components of a unit quaternion:

sensors\_event\_t.data[0] = rot\_axis.x\*sin(theta/2)

sensors\_event\_t.data[1] = rot\_axis.y\*sin(theta/2)

sensors\_event\_t.data[2] = rot\_axis.z\*sin(theta/2)

sensors\_event\_t.data[3] = cos(theta/2)

Where:

the x, y and z fields of rot\_axis are the East-North-Up coordinates of a unit length vector representing the rotation axis

theta is the rotation angle

The quaternion is a unit quaternion: it must be of norm 1. Failure to ensure this will cause erratic client behaviour.

In addition, this sensor reports an estimated heading accuracy:

sensors\_event\_t.data[4] = estimated\_accuracy (in radians)

The heading error must be less than estimated\_accuracy 95% of the time. This sensor must use a gyroscope as the main orientation change input.

This sensor also uses accelerometer and magnetometer input to make up for gyroscope drift, and it cannot be implemented using only the accelerometer and magnetometer.

Game rotation vector

Underlying physical sensors: Accelerometer and Gyroscope (no Magnetometer)

Reporting-mode: Continuous

getDefaultSensor(SENSOR\_TYPE\_GAME\_ROTATION\_VECTOR) returns a non-wake-up sensor

A game rotation vector sensor is similar to a rotation vector sensor but not using the geomagnetic field. Therefore the Y axis doesn't point north but instead to some other reference. That reference is allowed to drift by the same order of magnitude as the gyroscope drifts around the Z axis.

See the Rotation vector sensor for details on how to set sensors\_event\_t.data[0-3]. This sensor does not report an estimated heading accuracy: sensors\_event\_t.data[4] is reserved and should be set to 0.

In an ideal case, a phone rotated and returned to the same real-world orientation should report the same game rotation vector.

This sensor must be based on a gyroscope and an accelerometer. It cannot use magnetometer as an input, besides, indirectly, through estimation of the gyroscope bias.

Gravity

Underlying physical sensors: Accelerometer and (if present) Gyroscope (or magnetometer if gyroscope not present)

Reporting-mode: Continuous

getDefaultSensor(SENSOR\_TYPE\_GRAVITY) returns a non-wake-up sensor

A gravity sensor reports the direction and magnitude of gravity in the device's coordinates.

The gravity vector components are reported in m/s^2 in the x, y and z fields of sensors\_event\_t.acceleration.

When the device is at rest, the output of the gravity sensor should be identical to that of the accelerometer. On Earth, the magnitude is around 9.8 m/s^2.

If the device possesses a gyroscope, the gravity sensor must use the gyroscope and accelerometer as input.

If the device doesn’t possess a gyroscope, the gravity sensor must use the accelerometer and the magnetometer as input.

Geomagnetic rotation vector

Underlying physical sensors: Accelerometer and Magnetometer (no Gyroscope)

Reporting-mode: Continuous

Low-power

getDefaultSensor(SENSOR\_TYPE\_GEOMAGNETIC\_ROTATION\_VECTOR) returns a non-wake-up sensor

A geomagnetic rotation vector is similar to a rotation vector sensor but using a magnetometer and no gyroscope.

This sensor must be based on a magnetometer. It cannot be implemented using a gyroscope, and gyroscope input cannot be used by this sensor.

See the Rotation vector sensor for details on how to set sensors\_event\_t.data[0-4].

Just like for the rotation vector sensor, the heading error must be less than the estimated accuracy (sensors\_event\_t.data[4]) 95% of the time.

This sensor must be low power, so it has to be implemented in hardware.

Orientation (deprecated)

Underlying physical sensors: Accelerometer, Magnetometer and (if present) Gyroscope

Reporting-mode: Continuous

getDefaultSensor(SENSOR\_TYPE\_ORIENTATION) returns a non-wake-up sensor

Note: This is an older sensor type that has been deprecated in the Android SDK. It has been replaced by the rotation vector sensor, which is more clearly defined. Use the rotation vector sensor over the orientation sensor whenever possible.

An orientation sensor reports the attitude of the device. The measurements are reported in degrees in the x, y and z fields of sensors\_event\_t.orientation:

sensors\_event\_t.orientation.x: azimuth, the angle between the magnetic north direction and the Y axis, around the Z axis (0<=azimuth<360). 0=North, 90=East, 180=South, 270=West

sensors\_event\_t.orientation.y: pitch, rotation around X axis (-180<=pitch<=180), with positive values when the z-axis moves toward the y-axis.

sensors\_event\_t.orientation.z: roll, rotation around Y axis (-90<=roll<=90), with positive values when the x-axis moves towards the z-axis.

Please note, for historical reasons the roll angle is positive in the clockwise direction. (Mathematically speaking, it should be positive in the counter-clockwise direction):

Depiction of orientation

relative to a device

Figure 3. Orientation relative to a device.

This definition is different from yaw, pitch and roll used in aviation where the X axis is along the long side of the plane (tail to nose).

The orientation sensor also reports how accurate it expects its readings to be through sensors\_event\_t.orientation.status. See the SensorManager’s SENSOR\_STATUS\_\* constants for more information on possible values for this field.

Uncalibrated sensors

Uncalibrated sensors provide more raw results and may include some bias but also contain fewer "jumps" from corrections applied through calibration. Some applications may prefer these uncalibrated results as smoother and more reliable. For instance, if an application is attempting to conduct its own sensor fusion, introducing calibrations can actually distort results.

Gyroscope uncalibrated

Underlying physical sensor: Gyroscope

Reporting-mode: Continuous

getDefaultSensor(SENSOR\_TYPE\_GYROSCOPE\_UNCALIBRATED) returns a non-wake-up sensor

An uncalibrated gyroscope reports the rate of rotation around the sensor axes without applying bias compensation to them, along with a bias estimate. All values are in radians/second and are reported in the fields of sensors\_event\_t.uncalibrated\_gyro:

x\_uncalib: angular speed (w/o drift compensation) around the X axis

y\_uncalib: angular speed (w/o drift compensation) around the Y axis

z\_uncalib: angular speed (w/o drift compensation) around the Z axis

x\_bias: estimated drift around X axis

y\_bias: estimated drift around Y axis

z\_bias: estimated drift around Z axis

Conceptually, the uncalibrated measurement is the sum of the calibrated measurement and the bias estimate: \_uncalibrated = \_calibrated + \_bias.

The x/y/z\_bias values are expected to jump as soon as the estimate of the bias changes, and they should be stable the rest of the time.

See the definition of the gyroscope sensor for details on the coordinate system used.

Factory calibration and temperature compensation must be applied to the measurements. Also, gyroscope drift estimation must be implemented so that reasonable estimates can be reported in x\_bias, y\_bias and z\_bias. If the implementation is not able to estimate the drift, then this sensor must not be implemented.

If this sensor is present, then the corresponding Gyroscope sensor must also be present and both sensors must share the same sensor\_t.name and sensor\_t.vendor values.

Magnetic field uncalibrated

Underlying physical sensor: Magnetometer

Reporting-mode: Continuous

getDefaultSensor(SENSOR\_TYPE\_MAGNETIC\_FIELD\_UNCALIBRATED) returns a non-wake-up sensor

An uncalibrated magnetic field sensor reports the ambient magnetic field together with a hard iron calibration estimate. All values are in micro-Tesla (uT) and are reported in the fields of sensors\_event\_t.uncalibrated\_magnetic:

x\_uncalib: magnetic field (w/o hard-iron compensation) along the X axis

y\_uncalib: magnetic field (w/o hard-iron compensation) along the Y axis

z\_uncalib: magnetic field (w/o hard-iron compensation) along the Z axis

x\_bias: estimated hard-iron bias along the X axis

y\_bias: estimated hard-iron bias along the Y axis

z\_bias: estimated hard-iron bias along the Z axis

Conceptually, the uncalibrated measurement is the sum of the calibrated measurement and the bias estimate: \_uncalibrated = \_calibrated + \_bias.

The uncalibrated magnetometer allows higher level algorithms to handle bad hard iron estimation. The x/y/z\_bias values are expected to jump as soon as the estimate of the hard-iron changes, and they should be stable the rest of the time.

Soft-iron calibration and temperature compensation must be applied to the measurements. Also, hard-iron estimation must be implemented so that reasonable estimates can be reported in x\_bias, y\_bias and z\_bias. If the implementation is not able to estimate the bias, then this sensor must not be implemented.

If this sensor is present, then the corresponding magnetic field sensor must be present and both sensors must share the same sensor\_t.name and sensor\_t.vendor values.

Interaction composite sensors

Some sensors are mostly used to detect interactions with the user. We do not define how those sensors must be implemented, but they must be low power and it is the responsibility of the device manufacturer to verify their quality in terms of user experience.

Wake up gesture

Underlying physical sensors: Undefined (anything low power)

Reporting-mode: One-shot

Low-power

Implement only the wake-up version of this sensor.

getDefaultSensor(SENSOR\_TYPE\_WAKE\_GESTURE) returns a wake-up sensor

A wake up gesture sensor enables waking up the device based on a device specific motion. When this sensor triggers, the device behaves as if the power button was pressed, turning the screen on. This behavior (turning on the screen when this sensor triggers) might be deactivated by the user in the device settings. Changes in settings do not impact the behavior of the sensor: only whether the framework turns the screen on when it triggers. The actual gesture to be detected is not specified, and can be chosen by the manufacturer of the device.

This sensor must be low power, as it is likely to be activated 24/7.

Each sensor event reports 1 in sensors\_event\_t.data[0].

Pick up gesture

Underlying physical sensors: Undefined (anything low power)

Reporting-mode: One-shot

Low-power

Implement only the wake-up version of this sensor.

getDefaultSensor(SENSOR\_TYPE\_PICK\_UP\_GESTURE) returns a wake-up sensor

A pick-up gesture sensor triggers when the device is picked up regardless of wherever it was before (desk, pocket, bag).

Each sensor event reports 1 in sensors\_event\_t.data[0].

Glance gesture

Underlying physical sensors: Undefined (anything low power)

Reporting-mode: One-shot

Low-power

Implement only the wake-up version of this sensor.

getDefaultSensor(SENSOR\_TYPE\_GLANCE\_GESTURE) returns a wake-up sensor

A glance gesture sensor enables briefly turning the screen on to enable the user to glance content on screen based on a specific motion. When this sensor triggers, the device will turn the screen on momentarily to allow the user to glance notifications or other content while the device remains locked in a non-interactive state (dozing), then the screen will turn off again. This behavior (briefly turning on the screen when this sensor triggers) might be deactivated by the user in the device settings. Changes in settings do not impact the behavior of the sensor: only whether the framework briefly turns the screen on when it triggers. The actual gesture to be detected is not specified, and can be chosen by the manufacturer of the device.

This sensor must be low power, as it is likely to be activated 24/7. Each sensor event reports 1 in sensors\_event\_t.data[0].