Sensors HAL 1.0

Note: This page describes Sensors HAL 1.0. For information on implementing Sensors HAL 2.0 on devices running Android 10 and higher, see Sensors HAL 2.0 (/devices/sensors/sensors-hal2).

The Sensors HAL interface, declared in sensors.h
(https://android.googlesource.com/platform/hardware/libhardware/+/master/include/hardware/sensor
s.h)
, represents the interface between the Android framework
(/devices/sensors/sensor-stack#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://android.googlesource.com/platform/hardware/libhardware/+/master/include/hardware/sensor
s.h)
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 (#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 (#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 (/devices/sensors/batching#sampling_period_ns) and maximum report latency (/devices/sensors/batching#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 (/devices/sensors/batching#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 (/devices/sensors/batching#max_report_latency_ns) paragraph for more details.

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

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 (/devices/sensors/batching#sampling_period_ns).

flush(sensor)

```
int (*flush)(struct sensors_poll_device_1* dev, int sensor_handle);
```

Add a flush complete event (#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, `flush` must 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 (/devices/sensors/report-modes#one-shot) sensors; if `sensor_handle` refers to a one-shot sensor, `flush` must return `-EINVAL` and 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 (#get_sensors_list_list) function. See the definition of `sensors_module_t` in sensors.h

(https://android.googlesource.com/platform/hardware/libhardware/+/master/include/hardware/sensor s.h)
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 (/reference/hal/structhw__device__t)) defines the version number of the HAL.

sensor_t

`sensor_t` represents an Android sensor (/devices/sensors). 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? (/devices/sensors) for more details, and see Sensor types (/devices/sensors/sensor-types) 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://android.googlesource.com/platform/hardware/libhardware/+/master/include/hardware/sensor s.h)
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 (/devices/sensors/sensor-types#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 `maxRange` and 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 (/devices/sensors/sensor-types#base_sensors) for more details and see Power measurement process (/devices/sensors/power-use#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 (/devices/sensors/batching#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 (/devices/sensors/batching#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 (/devices/sensors/batching) for more details.

For sensors with an official sensor type, some of the fields are overwritten by the framework. For example, accelerometer (/devices/sensors/sensor-types#accelerometer) sensors are forced to have a continuous reporting mode, and heart rate (/devices/sensors/sensor-types#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 (#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://android.googlesource.com/platform/hardware/libhardware/+/master/include/hardware/sensor s.h)
and the definition of the different Sensor types (/devices/sensors/sensor-types) 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 (/devices/sensors/sensor-types) 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 (#flush_sensor) function for more information.