Sensors HAL 2.0

The Sensors Hardware Abstraction Layer (HAL) is the interface between the Android Sensors Framework and a device's sensors, such as an accelerometer or gyroscope. The Sensors HAL defines the functions that must be implemented to allow the framework to control the sensors.

Sensors HAL 2.0 is available in Android 10 and higher for new and upgraded devices. Sensors HAL 2.0 is based on Sensors HAL 1.0 (/devices/sensors/hal-interface) but has several key differences, which prevents it from being backwards compatible. Sensors HAL 2.0 uses Fast Message Queues (/devices/architecture/hidl/fmq) (FMQ) to send sensor events from the HAL into the Android Sensors Framework.

Examples and source

The main source of documentation for Sensors HAL 2.0 is within the HAL definition at hardware/interfaces/sensors/2.0/ISensors.hal (https://android.googlesource.com/platform/hardware/interfaces/+/master/sensors/2.0/ISensors.hal) . If there is a conflict of requirements between this page and `ISensors.hal`, use the requirement in `ISensors.hal`.

A default implementation is provided at hardware/interfaces/sensors/2.0/default (https://android.googlesource.com/platform/hardware/interfaces/+/master/sensors/2.0/default/). Unlike previous versions of the Sensors HAL, the default implementation is provided only as a reference and should not be shipped on any product. This implementation demonstrates the concept of using two FMQs to communicate with the framework.

Implementing Sensors HAL 2.0

To implement Sensors HAL 2.0, an object must extend the ISensors interface and implement all functions defined in `ISensors.hal`

(https://android.googlesource.com/platform/hardware/interfaces/+/master/sensors/2.0/ISensors.hal)
.

Initializing the HAL

The Sensors HAL must be initialized by the Android Sensors Framework before it can be used. The framework calls the `initialize()` function to provide three parameters to the Sensors HAL: two FMQ Descriptors and one pointer to an `ISensorsCallback` object.

The HAL uses the first descriptor to create the Event FMQ used to write sensor events to the framework. The HAL uses the second descriptor to create the Wake Lock FMQ used to synchronize when the HAL releases its wake lock for `WAKE_UP` sensor events. The HAL must save a pointer to the `ISensorsCallback` object so that any necessary callback functions may be invoked.

The `initialize()` function should be the first function called when initializing the Sensors HAL.

Exposing available sensors

To get a list of all of the available static sensors in the device, use the `getSensorsList()` function. This function returns a list of sensors, each uniquely identified by its handle. The handle for a given sensor must not change when the process hosting the Sensors HAL restarts. Handles may change across device reboots, and across System Server restarts.

If several sensors share the same sensor type and wake-up property, then the first sensor in the list is called the "default" sensor and is returned to applications that utilize the `getDefaultSensor(int sensorType, bool wakeUp)` API.

Stability of sensors list

After a Sensors HAL restart, if the data returned by `getSensorsList()` indicates a significant change compared to the sensor list retrieved before the restart, the framework triggers a restart of the Android Runtime. Significant changes to the sensor list include cases where a sensor with a given handle is missing or has changed attributes, or where new sensors are introduced. Although restarting the Android Runtime is disruptive to the user, it is required because the Android framework can no longer meet the Android API contract that static (non-dynamic) sensors do not change during the lifetime of an application. This may also prevent the framework from re-establishing active sensor requests made by applications. Therefore, HAL vendors are advised to prevent avoidable sensor list changes.
To ensure stable sensor handles, the HAL must deterministically map a given physical sensor in the device to its handle. Although no specific implementation is mandated by the Sensors HAL interface, developers have a number of options available to meet this requirement.

For example, the sensor list can be sorted using a combination of each sensor's fixed attributes, such as vendor, model, and sensor type. Another option relies on the fact that the device's set of static sensors is fixed in hardware, so the HAL needs to know when all expected sensors have completed initialization before returning from `getSensorsList()`. This list of expected sensors can be compiled into the HAL binary or stored in a configuration file in the filesystem, and the order of appearance can be used to derive stable handles. Although the best solution depends on your HAL's specific implementation details, the key requirement is that sensor handles do not change across HAL restarts.

Configuring sensors

Before a sensor is activated, the sensor must be configured with a sampling period and maximum reporting latency via the `batch()` function.

A sensor must be able to be reconfigured at any time via `batch()` without the loss of sensor data.

Sampling period

The sampling period has a different meaning based on the sensor type that is being configured:

- Continuous: The rate at which sensor events are generated
- On-Change: Events are generated no faster than the sampling period and may be generated at a rate slower than the sampling period if the measured value does not change.
- One-shot: Sampling period is ignored
- Special: For more details, see Sensor Type descriptions (/devices/sensors/sensor-types).

To learn about the interaction between sampling period and a sensor's reporting mode, see Reporting Modes (/devices/sensors/report-modes).

Maximum reporting latency

The maximum reporting latency sets the maximum time in nanoseconds, by which events can be delayed and stored in the hardware FIFO before being written to the Event FMQ 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 the sensor is present in the FIFO.

sensor is present in the FIFO.

For example, an accelerometer activated at 50Hz with a maximum reporting latency of 0 will trigger interrupts 50 times per second when the SoC is awake.

When the maximum reporting latency is greater than zero, sensor events do not need to be reported as soon as they are detected. Events can be temporarily stored in the hardware FIFO and reported in batches, as long as no event is delayed by more than the maximum reporting latency. 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 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 must 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 was reported.

For additional information and requirements on reporting sensor events with non-zero maximum reporting latency, see Batching (/devices/sensors/batching).

Activating sensors

The framework enables and disables sensors via the `activate()` function. Prior to activating a sensor, the framework must first configure the sensor via `batch()`.

Once a sensor is deactivated, additional sensor events from that sensor must not be written to the Event FMQ.

Flushing sensors

If a sensor is configured to batch sensor data, the framework can force an immediate flush of batched sensor events by calling `flush()`. This causes the batched sensor events for the specified sensor handle to be immediately written to the Event FMQ. The Sensors HAL must append a flush complete event to the end of the sensor events that are written as a result of a `flush()`.

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.

sensor. This applies to all sensors other than one-shot sensors.

If `flush()` is called for a one-shot sensor, then `flush()` must return BAD_VALUE and not generate a flush complete event.

Writing sensor events to the FMQ

The Event FMQ is used by the Sensors HAL to push sensor events into the Android Sensors framework.

The Event FMQ is a synchronized FMQ, which means that any attempt to write more events to the FMQ than the available space allows will result in a failed write. In such case, the HAL should determine whether to write the current set of events as two smaller groups of events or to write all of the events together when enough space is available.

When the Sensors HAL has written the desired number of sensor events to the Event FMQ, the Sensors HAL must notify the framework that events are ready by writing the `EventQueueFlagBits::READ_AND_PROCESS` bit to the Event FMQ's `EventFlag::wake` function. The EventFlag can be created from the Event FMQ via `EventFlag::createEventFlag` and the Event FMQ's `getEventFlagWord()` function.

Sensors HAL 2.0 supports both `write` and `writeBlocking` on the Event FMQ. The default implementation provides a reference for using `write`. If the `writeBlocking` function is used, the `readNotification` flag must be set to `EventQueueFlagBits::EVENTS_READ`, which is set by the framework when it reads events from the Event FMQ. The write notification flag must be set to `EventQueueFlagBits::READ_AND_PROCESS`, which notifies the framework that events have been written to the Event FMQ.

WAKE_UP events

`WAKE_UP` events are sensor events that cause the application processor (AP) to wake up and handle the event immediately. Whenever a `WAKE_UP` event is written to the Event FMQ, the Sensors HAL must secure a wake lock to ensure that the system stays awake until the framework can handle the event. Upon receiving a `WAKE_UP` event, the framework will

secure its own wake lock, allowing for the Sensors HAL to release its wake lock. To synchronize when the Sensors HAL releases its wake lock, use the Wake Lock FMQ.

The Sensors HAL must read the Wake Lock FMQ to determine the number of `WAKE_UP` events that the framework has handled. The HAL should only release its wake lock for `WAKE_UP` events if the total number of unhandled `WAKE_UP` events is zero. After handling sensor events, the framework counts the number of events that are marked as `WAKE_UP` sensor events, the framework counts the number of events that are marked as `WAKE_UP` events and writes this number back to the Wake Lock FMQ.

The framework will set the `WakeLockQueueFlagBits::DATA_WRITTEN` write notification on the Wake Lock FMQ whenever it writes data to the Wake Lock FMQ.

Dynamic sensors

Dynamic sensors are sensors that are not physically a part of the device but can be used as input to the device, such as a gamepad with an accelerometer.

When a dynamic sensor is connected, the `onDynamicSensorConnected` function in `ISensorsCallback` must be called from the Sensors HAL. This notifies the framework of the new dynamic sensor and allows the sensor to be controlled via the framework and to have the sensor's events be consumed by clients.

Similarly, when a dynamic sensor is disconnected, the `onDynamicSensorDisconnected` function in `ISensorsCallback` must be called so that the framework can remove any sensor that is no longer available.

Direct channel

Direct channel is a method of operation where sensor events are written to specific memory instead of to the Event FMQ bypassing the Android Sensors Framework. A client that registers a direct channel must read the sensor events directly from the memory that was used to create the direct channel and will not receive the sensor events via the framework. The `configDirectReport()`function is similar to `batch()` for normal operation and configures the direct report channel.

The `registerDirectChannel()` and `unregisterDirectChannel()` functions creates or destroys a new direct channel.

Operation modes

The `setOperationMode()`function allows for the framework to configure a sensor so that the framework can inject sensor data into the sensor. This is useful for testing, especially for algorithms that exist below the framework.

The `injectSensorData()` function is normally used to push operational parameters into the Sensors HAL. The function can also be used to inject sensor events into a specific sensor.

Validation

To validate your implementation of the Sensors HAL, run the sensor CTS and VTS tests.

CTS Tests

Sensor CTS tests exist in both automated CTS tests and the manual CTS Verifier application.

The automated tests are located in cts/tests/sensor/src/android/hardware/cts (https://android.googlesource.com/platform/cts/+/master/tests/sensor/src/android/hardware/cts). These tests verify standard functionality of sensors, such as activating sensors, batching, and sensor event rates.

The CTS Verifier tests are located in cts/apps/CtsVerifier/src/com/android/cts/verifier/sensors (https://android.googlesource.com/platform/cts/+/master/apps/CtsVerifier/src/com/android/cts/verifier/sensors/)
. These tests require manual input from the test operator and ensure that sensors report accurate values.

Passing the CTS tests is critical to ensuring that the device under test meets all CDD requirements.

VTS tests

VTS tests for Sensors HAL 2.0 are located in hardware/interfaces/sensors/2.0/vts (https://android.googlesource.com/platform/hardware/interfaces/+/master/sensors/2.0/vts/functiona)
. These tests ensure that the Sensors HAL is implemented properly and that all requirements within`ISensors.hal` and `ISensorsCallback.hal` are properly met.

Upgrading to Sensors HAL 2.0 from 1.0

When upgrading to Sensors HAL 2.0 from 1.0, ensure your HAL implementation meets the following requirements.

Initializing the HAL

The `initialize()` function must be supported to establish FMQs between the framework and HAL.

Exposing available sensors

In Sensors HAL 2.0, the `getSensorsList()` function must return the same value during a single device boot, even across Sensors HAL restarts. A new requirement of the `getSensorsList()` function is that it must return the same value during a single device boot, even across Sensors HAL restarts. This allows for the framework to attempt to re-establish sensor connections if the system server restarts. The value returned by `getSensorsList()` can change once the device performs a reboot.

Writing sensor events to the FMQ

Instead of waiting for `poll()` to be called, in Sensors HAL 2.0, the Sensors HAL must proactively write sensor events to the Event FMQ whenever sensor events are available. The HAL is also responsible for writing the correct bits to the EventFlag to cause an FMQ read within the framework.

WAKE_UP events

In Sensors HAL 1.0, the HAL was able to release its wake lock for any `WAKE_UP` event on any subsequent call to `poll()` after a `WAKE_UP` was posted to `poll()` because this indicates that the framework had processed all sensor events and had obtained a wake lock, if necessary. Because, in Sensors HAL 2.0, the HAL no longer knows when the framework has processed events written to the FMQ, the Wake Lock FMQ allows the framework to communicate to the HAL when it has handled `WAKE_UP` events.

In Sensors HAL 2.0, the wake lock secured by the Sensors HAL for `WAKE_UP` events must start with `SensorsHAL_WAKEUP`.

Dynamic sensors

Dynamic sensors were returned via the `poll()` function in Sensors HAL 1.0. Sensors HAL 2.0 requires that `onDynamicSensorsConnected` and `onDynamicSensorsDisconnected` in `ISensorsCallback` be called whenever dynamic sensor connections change. These callbacks are available as part of the `ISensorsCallback` pointer that is provided via the `initialize()` function.

`initialize()` function.

Operation modes

The `DATA_INJECTION` mode for `WAKE_UP` sensors must be supported in Sensors HAL 2.0.

Multi-HAL support

The Android Sensors Framework does not natively support Sensors HAL 2.0 with multi-HAL.