Qualcomm Linux Sensors Guide

80-70018-7 AA

April 8, 2025

Contents

1 Sensors overview

Note: The Qualcomm® sensing hub (QSH) is available only on QCS5430 and QCS6490.

The Qualcomm® system-on-chip (SoC) includes an application processor that runs the Linux operating system, a low-power application digital signal processor (aDSP), and other processors. The low-power processor runs the real-time operating system (RTOS) for handling the QSH use cases. The aDSP supports the following for QSH operations:

- GPIOs configurable as serial bus: serial peripheral interface (SPI), inter-integrated circuit (I^2C), improved I^2C (I^3C), and universal asynchronous receiver/transmitter (UART).
- Serial buses in low-power mode.
- Dedicated local memory, also known as the island in QSH.

1.1 QSH sensors

QSH provides a framework to use data from a wide range of sensors. The sensor data is useful in fields such as IoT, gaming, health, and fitness.

A device can have more than one sensor of a given type. For example, a flip-phone has an accelerometer placed on each of the two planes. The published attributes or capabilities distinguish each accelerometer sensor. You can access the availability, attributes, and capabilities of a sensor on the platform using the QSH client APIs. Use the same QSH client APIs to get the sensor data from the QSH framework.

The QSH framework APIs include QSH client APIs and sensor APIs, to perform the following sensor-related tasks:

- Identify the sensors available on a development kit.
- Determine sensor capabilities using attributes, such as supported sample rate, maximum range, manufacturer, power requirement, and resolution.
- Collect and provide data according to the configuration, thereby enabling sensors with a specified sample rate.

The QSH framework provides access to both hardware-based and software-based sensors.

Hardware-based sensors

Hardware-based sensors are physical sensors that gather data by directly measuring specific environmental properties, such as acceleration, magnetic field, pressure, humidity, light, and angular velocity.

The following table lists the hardware-based sensors that the QSH framework supports:

Table : Hardware-based sensors

Sensor name	Sensor type	Description	Proto API
Accelerometer	`accel`	Measures the acceleration applied to a device on all the 3 physical axes (x, y, and z) in meter/second square (m/s2)	`sns_accel.proto`
Gyroscope	`gyro`	Measures the rate of rotation of a device around each of the 3 physical axes (x, y, and z) in radians/second (rad/s)	`sns_gyro.proto`
Sensor temperature	`sensor_temperature`	Measures the temperature of the sensor in degrees Celsius (℃)	`sns_sensor_temperature.proto`
Magnetometer	`mag`	Measures the ambient magnetic field for all the 3 physical axes (x, y, and z) in microtesla (T)	`sns_mag.proto`
Proximity	`proximity`	Measures the proximity of an object and provides *near/far* events	`sns_proximity.proto`

Sensor name	Sensor type	Description	Proto API
Ambient light	`ambient_light`	Measures the ambient light level illumination in lux (lx)	`sns_ambient_light.proto`
Pressure	`pressure`	Measures the ambient air pressure in hectoPascal (hPa)	`sns_pressure.proto`
Humidity	`humidity`	Measures the relative ambient humidity in percentage (%)	`sns_humidity.proto`
Ambient temperature	`ambient_temperature`	Provides the ambient room temperature in degrees Celsius (℃)	`sns_ambient_temperature.proto`
Hall	`hall`	Measures the magnetic field and provides a magnet *near/far* indication	`sns_hall.proto`
Capacitive proximity	`sar`	Detects human object proximity using change in capacitance and reports *near/ far* events	`sns_sar.proto`

Software-based sensors

Software-based sensors, also known as virtual sensors, are algorithms that gather data from one or more physical sensors and generate the intended output. The common examples are gravity, step counter, and game rotation vector.

The following table lists the software-based sensors that the QSH framework supports:

Table : Software-based sensors

Sensor name	Sensor type	Proto API	Description
Absolute motion detector	`amd`	`sns_amd.proto`	• Reports a stationary state event when the device is at absolute rest. For example, the device is placed on a stationary object, such as desk or table. • Reports a moving state event when the device transitions from absolute rest to moving state. For example, the device is being lifted from a desk or table. • Uses the accelerometer motion detect interrupt to reduce the power.
Relative motion detector	`rmd`	`sns_rmd.proto`	Reports a stationary state when the device is not moving significantly with respect to gravity.

Sensor name	Sensor type	Proto API	Description
Significant motion detector	`sig_motion`	`sns_sig_motion.proto`	• Triggers when detecting a significant motion - a motion that might lead to a change in the user location. For example, walking, biking, or sitting in a moving car, coach, or train. • The following examples do not trigger a significant motion: – The device is in a pocket and the person is not moving. – The device is on a table and the table shakes a bit. • Reporting mode: Single response, after the notification sensor automatically disables itself.
Pedometer	`pedometer`	`sns_pedometer.proto`	Reports the number of step counts to the client.
Step detector	`step_detect`	`sns_step_detect.proto`	Detects steps and generates an event on each step.

Sensor name	Sensor type	Proto API	Description
Tilt detector	`tilt`	`sns_tilt.proto`	Generates an event each time there is a tilt. The direction of the 2-second window, with average gravity changing by at least 35 degrees since the activation or the last event generated by the sensor, defines a tilt event.
Tilt to wake	`tilt_to_wake`	`sns_tilt_to_wake.proto`	Detects the substantial device rotation gesture event when the picked device is in a specific range of the pitch and roll angles.
Gyroscope calibration	`gyro_cal`	`sns_gyro_cal.proto`	• A low-power dynamic calibration algorithm for gyroscopes. • Validated across multiple gyroscope parts from different vendors.

Sensor name	Sensor type	Proto API	Description
Magnetometer calibration	mag_cal	sns_mag_cal.proto	• A low-power dynamic calibration algorithm for the magnetometer sensor. • Validated across multiple magnetometer parts from different vendors.
Game rotation vector	game_rv	sns_game_rv.proto	• Reports the orientation of the device that is relative to an unspecified coordinate frame. • Obtains the orientation through integration of accelerometer and gyroscope readings. Therefore, the Y-axis does not point north; instead, it points to an arbitrary reference.

Sensor name	Sensor type	Proto API	Description
Gravity/linear acceleration	`gravity`	`sns_gravity.proto`	• Provides a three-dimensional vector indicating the direction and magnitude of gravity. • Determines the relative orientation of the device in space.
Persistent stationary detector	`persist_stationary_detect`	`sns_persist_stationary_detect.proto`	Reports an event when the device is stationary for at least 5 seconds.
Persistent motion detector	`persist_motion_detect`	`sns_persist_motion_detect.proto`	Reports an event when the device is in motion for at least 5 seconds.
Device orientation	`device_orient`	`sns_device_orient.proto`	Reports whether the device is in portrait mode or landscape mode.
Geo-mag rotation vector (RV)	`geomag_rv`	`sns_geomag_rv.proto`	Reports the orientation of the device relative to the East-North-Up coordinates frame; obtained through the integration of accelerometer and magnetometer readings.

Sensor name	**Sensor type**	**Proto API**	**Description**
Rotation vector	`rotv`	`sns_rotv.proto`	• Reports the orientation of the device relative to the East-North-Up coordinates frame. • Obtains orientation through the integration of accelerometer, gyroscope, and magnetometer readings.
Device position classifier	`device_position_classi fier`	`sns_dpc.proto`	Provides the device position information.
Activity recognition algorithm	`activity_recognition`	`sns_activity_recognition.proto`	Determines relative stationary, such as walk, run, bike, car, nonmotorized vehicle, and motorized vehicle states and classifications.

Sensor name	Sensor type	Proto API	Description
Distance bound	distance_bound	sns_distance_bound.proto	• Tracks the distance in meters, and reports to the client when the requested distance is covered. • The client can query the accumulated distance anytime before the final distance is reached.

Feature	Enable or disable?	Impact	Description

Features

Describes the Qualcomm® sensing hub (QSH) features and their impact on the IoT use cases.

The following table describes the impact on QSH for deployment in scenarios with IoT devices:

Table : QSH features

Feature	Enable or disable?	Impact	Description
OS and hardware-independent	N.A.	• Development is easy. • Plug and play of QSH-compliant sensors across the targets.	• QSH drivers or algorithms are compliant across Qualcomm® Snapdragon platforms. • APIs are generic, and abstract the underlying OS.
Sensors in local memory	Yes.	• Saves power by enabling the local memory. • Due to the limited local memory, only finite sensors fit in here.	• Enables QSH to operate in ultralow power mode, meeting the low-power requirements of the industry. • Suitable for background use cases.

Feature	Enable or disable?	Impact	Description
Sensors in Normal mode	Yes.	• More memory and more on-chip resources are available. • Higher power consumption due to more on-chip resources being active.	• Access to the larger main memory allows integration of more sensors. • Supports cycle-intensive operations.
Factory calibration	No.	• Improved sensor accuracy.	• By default, this feature is available. • QSH supports calibration for hardware-based sensors. • Calibration standardizes equipment for precise results, which ensures that the sensor values match the baseline. • Recalibration maintains sensor accuracy and adjusts for sensitivity changes over time. • For more information, see Calibrate sensors.

Feature	Enable or disable?	Impact	Description
Flexibility in configuring the sensors	No.	• Configure the buses, Power rails, GPIOs, Interrupt/Polling modes and so on, for physical sensors.	• By default, this feature is available. • Change sensors configuration, such as Serial Bus type, GPIOs, and Interrupt/Polling mode. • Sensors are configured using the Registry files parsed during QSH framework initialization. • For more information, see Configure sensors in Qualcomm Linux Sensors Guide - Addendum.

Feature	Enable or disable?	Impact	Description
Software-based sensors	Yes.	• Includes device motion, activity, and device physical position sensors.	• These sensors are a set of software algorithms. • Enables device motion using significant motion. • Supports device positioning and direction using linear acceleration, gravity, geomagnetic rotation vector, and rotation vector. • Step counter, step detector, and activity recognition help identify user activities. • For more information, see Software-based sensors.

Feature	Enable or disable?	Impact	Description
QSH direct channel	Yes.	• Low latency and improved performance.	• The QSH direct channel is an interface designed for high-speed applications. It ensures that sensor data is transmitted with minimal delay, which is crucial for high-rate applications that require real-time or near-real-time data processing. • The overall system performance is enhanced due to the efficient handling and quick access to sensor data, making it suitable for applications that demand high-speed data processing. • For more information, see QSH direct channel APIs.

QSH architecture

Describes the QSH unified event-driven framework.

Note:

- To continue reading about the APIs or set up sensor information, see QSH APIs.
- Source code of the low-power application digital signal processor (aDSP), including the QSH framework, is available only to licensed users with authorized access. To upgrade your access, go to: www.qualcomm.com/support/working-with-qualcomm.

QSH, also known as Qualcomm® Snapdragon™ sensor core (SSC), offers a unified event-driven framework for drivers and algorithms. QSH supports the same set of APIs for both the hardware-based and software-based sensors. Additionally, QSH supports asynchronous bus transfer and is extendable for new or custom driver features. QSH consists of the following components:

- QSH client APIs
- Sensor APIs
- Core framework
- Pre-implemented platform sensors
- Vendor-implemented sensors
- Test modules

QSH serves external clients and provides a simple interface to access sensor data. The following table describes the terms used in the QSH framework:

Table : QSH terminology

Item	Description
Sensor	• Produces a single type of data; for example, accelerometer, gyroscope, timer, interrupt, and rotation vector • Handles asynchronous data • Publishes mandatory and custom attributes, and manages its instances

Item	Description
Sensor instance	• Runs at a specific configuration, publishes output data events, and can be created per client request or shared among multiple requests • Physical sensors usually share a single instance
Sensor unique identifier (SUID)	A unique 128-bit ID for each sensor
Service	A module that provides a synchronous interface for common utilities
Data stream	A unique connection between a client and data source
Request	A configuration message that a client sends to a sensor (see `sns_request.h` file)
Event	Asynchronous output data message that a sensor instance generates (see `sns_sensor_event.h` file)
Nanopb	A small code-size protocol buffer implemented in ANSI C

The following figure shows the QSH architecture:

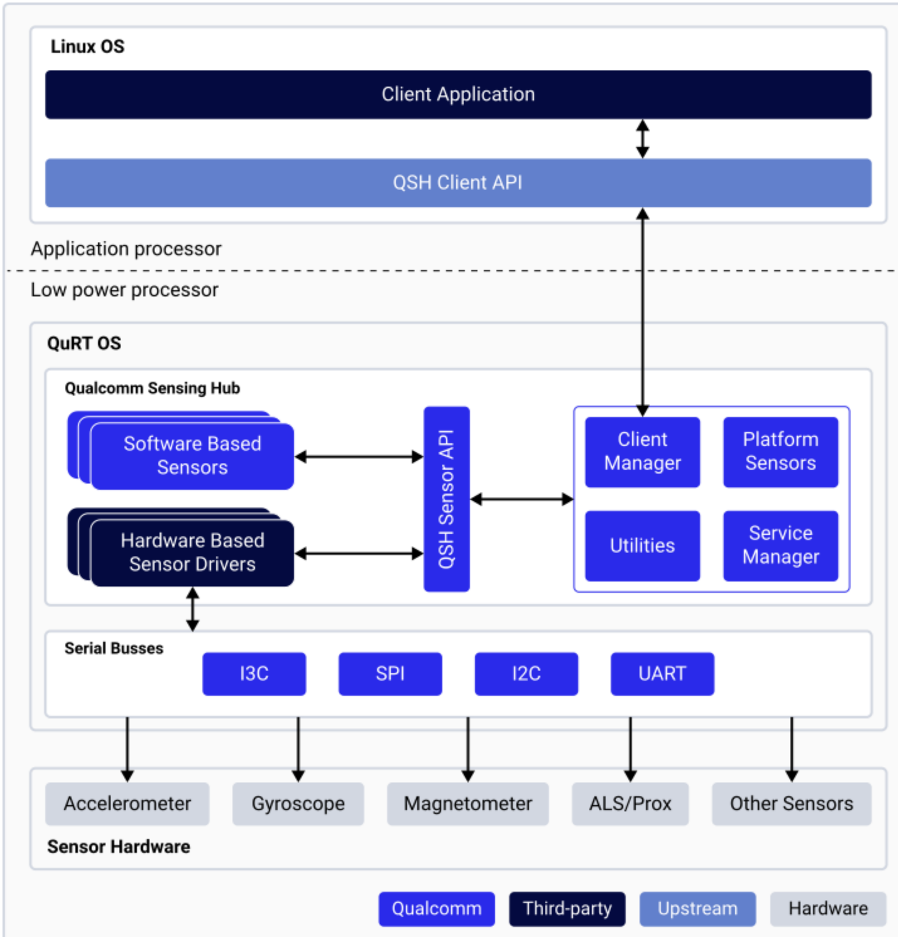

Figure: QSH architecture

The QSH framework includes the following components:

- **Application processor software modules**
  - **Client application**: It has the *application main()* or *entry function* that interacts with the QSH client APIs on the application processor side.
  - **QSH client APIs**: It offers high-level APIs to access services offered by QSH. It simplifies application development by abstracting system complexities and focusing on the application logic. For more information, see QSH APIs.
- **Low-power processor software modules**
  - **Client manager**: The client manager is in charge of all communications of the low-power processor with the application processor. It's responsible for the following:

Table : Client manager functions

Function	Description
Translate incoming requests	The client manager takes incoming requests and translates them into a format that the QSH can understand.
Translate outgoing indications	The client manager receives event messages from the QSH and translates these event messages into outgoing indications in a format that is understandable outside the QSH.
Guarantees batching options	If a client specifies certain batching (store/accumulate locally) options, the client manager ensures that they meet the batching options. The client manager checks that the data is grouped and sent in the same way as the client has specified, ensuring compliance with the criteria.

– **Service manager**: QSH offers synchronous services through its service manager. The sensor and sensor instance APIs use a callback to connect to this service manager.

  The `adsp_proc/qsh_platform/inc/sns_service.h` file lists the services available in the QSH, also referred as QSH services. The following table describes the key QSH services that are essential for device drivers.

Table : QSH services

QSH service	Description
Stream service	◦ Allows creating and removing a data stream with a sensor. ◦ See the `adsp_proc/qsh_platform/inc/sns_data_stream.h` file for data stream API to send requests and receive events over the data streams.

QSH service	Description
Attribute service	○ Allows a sensor to publish sensor attributes or capabilities. ○ All standard attribute IDs and expected value type are defined in the `sns_std_sensor.proto` file. ○ All attribute values must be in the nanopb-encoded format. ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_attribute_service.h` file.
Diagnostic service	○ Provides debug message and data log packet services, and defines standard log packet IDs. ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_diag_service.h` file.
Event service	○ Allows to publish output events from source sensor instances. ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_event_service.h` file.
Power rail service	○ Available to the physical sensors to register and vote for the power rails (ON/OFF). ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_pwr_rail_service.h` file.

QSH service	Description
Synchronous COM port (SCP) service	○ Available to the physical sensors to register/deregister the COM port and perform synchronous transfers over the COM port. ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_sync_com_port_service.h` file.
GPIO service	○ Available to the physical sensors to read/write the GPIO value. ○ Effectively abstracts a low-level CoreBSP layer for controlling the GPIOs. ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_gpio_service.h` file.
Island service	○ Available to the physical sensors to request for island exit. ○ When an application must access DDR or nonisland resources, the application code can use the island service. ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_island_service.h` file.
File system service	○ Available to the physical sensors for the file service management. ○ The abstract file system is a part of an application processor stack and can be available for local access from a low-power processor. ○ For more information about the API, see the `adsp_proc/qsh_platform/inc/services/sns_file_service.h` file.

– **Platform sensor**: QSH provides certain built-in sensors for platform or hardware-specific abstraction that other sensors and sensor instances can use. The following table describes the platform sensors:

Table : Platform sensors

Platform sensor	Description
Registry sensor	◦ The registry sensor in QSH provides an interface for sensors to access registry data from persistent memory. It allows sensors to create a data stream, send requests, receive data events, subscribe to updates, and remove unnecessary data streams. ◦ For more information about the registry sensor, see Configure sensors in Qualcomm Linux Sensors Guide - Addendum. **Note:** Qualcomm Linux Sensors Guide - Addendum is available to licensed developers with authorized access. ◦ The registry sensor API is documented in the `adsp_proc/qsh_api/pb/sns_registry.proto` file.
Timer sensor	◦ The timer sensor in the QSH offers an interface to initiate periodic or one-shot timers. Sensors that require timers must create a data stream, send requests, and read delivered data events. ◦ The timer sensor API is documented in the `adsp_proc/qsh_platform/api/public_sns/sns_timer.proto` file.
Interrupt sensor	◦ The interrupt sensor in the QSH offers an interface to register interrupts. Sensors that require interrupts must create a data stream, send requests, and read delivered data events. ◦ The interrupt sensor API is documented in the `adsp_proc/qsh_platform/api/public_sns/sns_interrupt.proto` file.

<table>
<tr><th>Platform sensor</th><th>Description</th></tr>
<tr><td>Asynchronous COM port (ASCP) sensor</td><td>

The ASCP sensor in the QSH offers an interface for asynchronous read and write operations over a communication port.
Sensors that require this feature must create a data stream, send requests, and read delivered data events.
The ASCP sensor API is documented in the <code>adsp_proc/qsh_platform/api/public_sns/sns_async_com_port.proto</code> file.

Note: The ASCP sensor is typically used by physical sensor drivers to read large FIFO.
</td></tr>
<tr><td>SUID lookup sensor</td><td>

The SUID lookup sensor in the QSH provides an API to obtain the SUID of dependent sensors. Its own SUID is available using the <code>sns_get_suid_lookup()</code> function in the <code>sns_sensor_util.h</code> file.
The <code>adsp_proc/qsh_api/pb/sns_suid.proto</code> file documents the SUID lookup sensor API.

</td></tr>
<tr><td>Test sensor</td><td>

The test sensor customizes and runs sensor-specific use cases.
The test sensor is available in the <code>adsp_proc/qsh_platform/sensors/test</code> directory.

</td></tr>
</table>

– **QSH utilities**: QSH provides several helper utilities for sensors and sensor instances. All the utilities are available in the `adsp_proc/qsh_platform/inc/utils` directory. The following table describes the key utilities:

Table : QSH utilities

QSH utility	Description
Nanopb encode/decode	○ Provides common encode/decode helper functions for all the sensors. For example, encode/decode `sns_request` messages, encode and publish/decode data events. ○ Asynchronous COM port nanopb utilities are available for physical sensor drivers.
Sensor utils	○ Provides common functionalities, such as finding a sensor instance and getting the SUID of a SUID lookup sensor.
Attribute utils	○ Provides helper functions that encode and publish a sensor attribute.
Memory utils	○ Provides helper functions for efficient memory management and allocation.
Math utils	○ Offers a collection of mathematical functions and operations such as matrix, FFT, and IIR filter.
Printf utils	○ Includes helper functions to format and print data.

Sensor and sensor instances

QSH divides the sensor implementation in two logical units: sensor and sensor instance.

- Sensors are producers or consumers, or a combination of producers and consumers of asynchronous data.
  - Each sensor can have one or more sensor instances.
  - Any request to a sensor for data results in the creation of a sensor instance or sharing of an existing sensor instance.
- The sensor creates sensor instances on demand.
  - Sensors manage the lifecycle and configuration of their corresponding instances, and sends configuration updates and initial state events to their clients.
  - Each sensor instance operates with a specific client configuration.
  - The sensor instance of a physical sensor programs the sensor hardware to operate at required configuration.
  - Vendors must serve all client requests with a minimal number of sensor instances.
  - The sensor instance generates and sends a stream of data to all the active clients.
- Many sensors can share and configure a single sensor instance - this mode of operation is typical to a combo driver for hardware sensors, such as:
  - Accelerometer and gyroscope
  - Proximity and ambient light

Communication among sensors

Every algorithm and sensor driver within the QSH framework is called a sensor, with the standard QSH APIs. Information exchange across these sensors is necessary for any real use case.

All communication to, from, and among the sensors is performed through the request and event messages over data streams. The message payloads are defined in the protocol buffer format, using the nanopb generator, encoder, and decoder. The message payload length, message ID, and timestamp (for events) are communicated within the metadata managed by the QSH framework.

The following figure shows the communication between the data client and the data source, using the data stream:

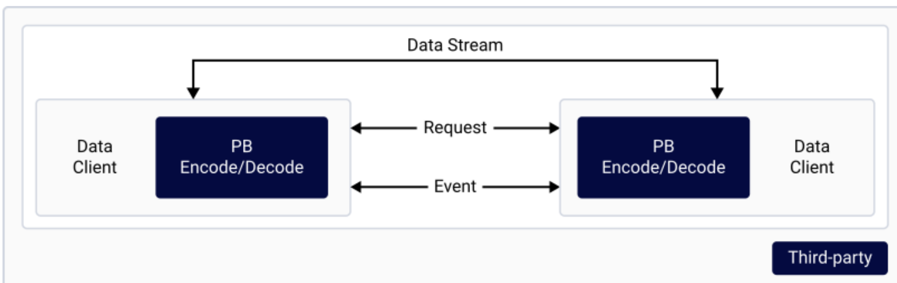

Figure: Sensor communication between client and source

- The client sends request messages to enable, disable, and reconfigure a sensor. Request messages are always addressed to a specific SUID. After the target sensor receives the request message, it sends the request to the sensor instance for proper handling.
- Sensor instances send event messages asynchronously to their registered clients, which can be other sensors or sensor instances.

Nanopb protocol buffer in QSH

The QSH uses nanopb protocol buffer for the following reasons:

- All the request and event messages that are exchanged between the sensors.
  - A sensor or sensor instance must encode the payload (if present) for all requests it sends to its dependents.
  - A sensor or sensor instance must decode the payload (if present) for all requests it receives.
  - A sensor or sensor instance must encode the payload (if present) for all events it publishes.
  - A sensor or sensor instance must decode the payload (if present) for all events it receives from its dependents.

Note: Certain requests or events don't have a message body. In this case, decoding or encoding the payload isn't expected, and the sensor processes these messages based on their message ID.

- Representing the attribute data
  - All attribute values are in the nanopb-encoded format.
- Diagnostic log packet payload
  - All payloads in the diagnostic log packets are in the nanopb-encoded format.

Note: For more information about protocol buffers, see Protocol-buffers and nanopb.

File	Description

Sensor API messages

The following files refer the API messages, which contain the message definitions, and allow communication between sensors:

- The `.proto` files contain the protocol buffer message definitions and documentation that allow communication between sensors.
- The following table lists the API standard message defined in the `<workspace>/build-qcom-wayland/workspace/sources/sensinghub/sensing-hub/apis/proto/sns_std_*.proto` file:

Table : Standard proto files

File	Description
`sns_std.proto`	This file includes standard definitions, such as: – Message ID – Request message – Batching specification – An attribute request and event – An error event
`sns_std_sensor.proto`	This file includes definitions, such as: – Message IDs for request and event APIs of standard sensors – Streaming and event messages – Sensor sample status types – Standard attribute IDs – Common attribute types – A physical sensor configuration event message
`sns_std_type.proto`	This file includes common API-type definitions, such as: – SUID messages – Attribute events and value messages – Common error types
`sns_std_event_gated_sensor.proto`	This file includes the API for event gated sensors, encompassing the configuration message ID and API documentation.

- Physical sensor-specific API definitions and documentation are present in the sensor-specific `.proto` files. For example, `sns_accel.proto`, `sns_proximity.proto` and `sns_` `motion_detect.proto`.

- The QSH platform sensor API definitions and documentation are present in the `adsp_proc/qsh_platform/api/` directory. For example, `sns_timer.proto`, `sns_interrupt.proto`, and `sns_async_com_port.proto`.
- The following proto files defines the framework-related APIs for SUID, registry, and diagnostics:
  - `sns_suid.proto`
  - `sns_registry.proto`
  - `sns_diag.proto`

Next steps

- QSH APIs

QSH APIs

Describes the QSH interfaces and important QSH functions, classes, methods, and data structures.

The QSH interfaces are available for working with any kind of hardware-based, software-based, pre-existing, or any other QSH-compliant sensors.

The QSH framework runs on the low-power processors and exposes the APIs at application processor. These APIs include the following:

- QSH client APIs and various feature APIs for application development.
- Sensor APIs for creating new sensors on a low-power processor.

Application processor APIs

QSH offers various feature APIs at the application processor, to meet specific needs of different use cases.

QSH client APIs

The QSH client APIs provide a simple and easy interface to the client code that allows the QSH functionality and develops an end-to-end sensor use case. The QSH supports the following APIs:

getSession()

This API creates an instance of `ISession` and returns a pointer to it.

Syntax

```
ISession* getSession();
```

Parameters

None.

Response

Return	Description
`ISession*`	`ISession`: Upon success `nullptr`: Upon failure

Open()

This API initiates the client session created using `getSession()`. It sets up and establishes communication between the client and the QSH framework. The client must call this function only once per session.

Syntax

```
int open();
```

Parameters

None.

Response

Return	Description
`int`	`0`: Successfully opened session `-1`: Upon failure

setCallBacks()

For a specified SUID, this API allows the user to set callbacks for responses, errors, and events received over the session opened using `open()`. This API registers the SUID with the callback functions passed as parameters to this API.

To unset the callbacks for already registered SUID, the user can pass `nullptr` for all the three callback functions - `respCallBack`, `errorCallBack`, and `eventCallBack`. Passing `nullptr` for all callback functions results in an error for any unregistered SUID.

Syntax

```
virtual int setCallBacks(suid suid, respCallBack respCB,
errorCallBack errorCB, eventCallBack eventCB) = 0;
```

Parameters

Parameter	Name	Description
input	suid	Unique SUID of the sensor for which the callbacks are being set.
input	respCB	Response callback function.
input	errorCB	Error callback function.
input	eventCB	Event callback function.

Note: If any callback function isn't defined or required, then the client can pass `nullptr`.

Response

Return	Description
int	0: Upon success -1: Upon failure — if all the callback functions are `nullptr`, for an unregistered SUID.

sendRequest()

This API asynchronously sends a protocol buffer (proto) encoded message to the QSH framework. The proto-encoded message has instructions or configuration settings for the sensor identified by the SUID.

If the user sets callback pointers using `setCallBacks()` for the specified SUID, then the response, event, or a combination of responses and events are received from the QSH framework.

Syntax

```
virtual int sendRequest(suid suid, string message) = 0;
```

Parameters

Parameter	Name	Description
input	suid	A unique SUID of the sensor for request is to be sent.
input	message	Proto-encoded request message to be sent.

Response

Return	Description
`int`	`0`: Upon success `-1`: Upon failure, due to: • User tries to send a request over a closed session • Encoded message size exceeds the permissible size • Sending message fails due to the channel related issue

Close()

This API closes the sensor session and terminates the communication between the client and the QSH framework. The client must call this function at the end of the session to release the resources and avoid memory leaks.

Syntax

```
void close();
```

Parameters

None.

Response

Return	Description
`void`	None

For more information and example codes, see sample1.

QSH direct channel APIs

QSH direct channel APIs provides low latency sensor data channel for high rate applications. Direct channel APIs use the fastRPC interface, which bypasses the standard interprocessor communication (IPC) mechanism to achieve faster communication. For more information, see QSH APIs in Qualcomm Linux Sensors Guide - Addendum.

Note: Qualcomm Linux Sensors Guide - Addendum is available to licensed users with authorized access.

Low-power processor APIs

The low-power processor APIs provide access to:

- QSH framework resources, such as services and utilities
- Core functionalities, such as the diagnostic interface and timer

You can use the low-power processor APIs to create algorithms and sensor drivers that comply with the QSH standards. This information is available to the licensed users who have full access to the proprietary software shipped with Qualcomm Linux. For more information, see QSH APIs in Qualcomm Linux Sensors Guide - Addendum.

Note: Qualcomm Linux Sensors Guide - Addendum is available to licensed users with authorized access.

Software

Describes the application processor and low-power processor directory structure.

To design and manage the APIs, it's necessary to understand the organization of APIs in the code and the relationships, functions, and operations of the source code directory files.

Application processor directory structure

The QSH-exposed APIs are called QSH client APIs for application development. The following information shows the QSH client API tree structure and the location of each software component.

The source code for QSH client APIs is in `<workspace>/build-qcom-wayland/workspace/sources/sensinghub/sensing-hub` directory.

```
├───/sources/sensinghub/sensing-hub
│
├───apis                            - APIs
│   └───proto                       - Proto API
│       └───google
│           └───protobuf
│
├───common                          - Common header file
│   └───inc
│
├───examples
│   └───SessionClient               - Sample 1 source file
│
└───session                         - Client APIs source and header
```

```
files
    └───1.0
        ├───inc
        └───src
```

Note: To view and customize the source code, see the Qualcomm Linux Yocto Guide. The sensing-hub source is available at `<workspace>/build-qcom-wayland/workspace/sources/sensinghub/sensing-hub/`.

Low-power processor directory structure

The QSH software stack consists of the framework, API, sensor driver, and algorithm components. This information is available to licensed users with authorized access. For more information, see Software in Qualcomm Linux Sensors Guide - Addendum.

Note: Qualcomm Linux Sensors Guide - Addendum is accessible only to the licensed users with authorized access.

Next steps

- Bring up sensors

Platform

Describes the sensors supported on Qualcomm® RB3 Gen 2 Development Kit.

The Qualcomm® RB3 Gen 2 Development Kit includes an application processor and the Hexagon DSP, which is also called as the low-power processor. The QSH resides on the low-power processor that runs with Qualcomm Real Time (QuRT™) OS. The QSH client API resides on the application processor that runs with Linux OS. The QuRT OS is designed for the Hexagon DSP.

Note: See Hardware SoCs that are supported on Qualcomm Linux.

RB3 Gen 2 Development Kit

The RB3 Gen 2 Development Kit includes the following hardware, sensor parts, and connectivity configurations.

Hardware specification for low-power processor

- CPU clock up to 1.4 GHz Turbo
- Low-power island
- 2 MB local memory
  - 1 MB reserved for QSH and relevant CoreBSP dependencies
  - 1 MB reserved for audio and relevant CoreBSP dependencies
- Five dedicated buses for sensors: one I^3C, one SPI, one I^2C, and two UARTs
- Twelve dedicated GPIOs with local memory for sensors

For more information about the low-power processor, see RB3 Gen 2 Development Kit.

Sensors on the platform

The following table lists the sensors supported on the RB3 Gen 2 Development Kit:

Table : Core Kit sensor

Placement	Sensor part	Sensor type
Main board	ICM42688	Accelerometer and gyroscope

Table : Vision Kit sensor

Placement	Sensor part	Sensor type
Main board	ICM42688	Accelerometer and gyroscope
Vision mezzanine	ICM42688	Accelerometer and gyroscope
	ICP-10111	Pressure/barometer
	AK09915	Magnetometer

Note: On the Vision kit, only one accelerometer and gyroscope sensor part can be enabled at a time.

Sensors and serial bus configuration

The following table lists the GPIO numbers, serial bus, QSH registry configuration, and interrupts for different sensors on the RB3 Gen 2 Development Kit:

Table : Serial bus configuration

GPIOs	Interface		Sensors connected	QSH registry parameters	Interrupts

GPIO_159
GPIO_160
	I^2C	Magnetometer (AK09915)
		Pressure (ICP-10111)
			`bus_type: 0 (SNS_BUS_I2C)`
			`bus_instance: 1`
			`max_bus_speed_khz: 400`
			`slave_config:` : I^2C slave address
				Polling
				Note: Pressure and magnetometer sensors are configured in the Polling mode in the RB3 Gen 2 Development Kit.	

GPIO_161
GPIO_162
	I^3C	Not connected	`bus_type: 3 (SNS_BUS_I3C_SDR)`
			`bus_instance: 2`
			`max_bus_speed_khz: 12500`
			`slave_config::` $I^2$2C static address
			`i3c_address:` User-defined I3C dynamic address
				NA	

GPIOs	Interface		Sensors connected	QSH registry parameters	Interrupts

GPIO_163
GPIO_164
GPIO_165
GPIO_166
	SPI	Accelerometer and gyroscope (ICM42688)	`bus_type: 1 (SNS_BUS_SPI)`
			`bus_instance: 3`
			`max_bus_speed_khz: 9600`
			`slave_config: 0`: For SPI, this value indicates the chip-select line for the slave.
				GPIO_103	

GPIO_171 GPIO_172	UART	BTLE	`bus_type: 2 (SNS_BUS_UART)` `bus_instance: 36`	NA	
GPIO_173 GPIO_174	UART	Debug	`bus_type: 2 (SNS_BUS_UART)` `bus_instance: 7`	NA	

Enable sensors on the RB3 Gen 2 Development Kit

Use the following resources to set up and validate the sensors based on either the Core kit or the Vision kit.

Enable sensor on the Core Kit

Hardware set up: The following resources cover the hardware set up to enable the sensor on the main board.

Table : Main board DIP switch (DIP_SW_0) setting

DIP switch number	State	Purpose
5	OFF	Enable the sensor on the main board.

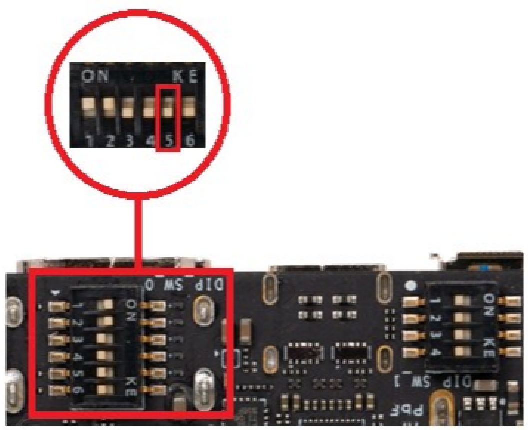

Figure: DIP switch setting to disable accelerometer and gyroscope sensor on main board

Software set up: No software change is required to enable the accelerometer and gyroscope sensor on the Core kit.

Enable sensor on the Vision Kit

Hardware set up: The following resources cover the hardware setup to enable sensors on the vision mezzanine board.

Table : Main board DIP switch (DIP_SW_0) setting

DIP switch number	State	Purpose
5	ON	Disable the sensor on the main board.

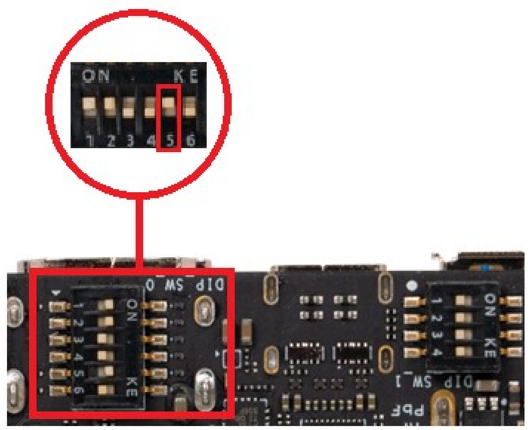

Figure: DIP switch setting to disable accelerometer and gyroscope sensor on main board

Table : Vision mezzanine DIP switch (DIP1) setting

DIP switch number	State	Purpose
1	ON	Enable the accelerometer and gyroscope sensor.
4	ON	Enable the pressure sensor.
5	ON	Enable the magnetometer sensor.

Table : Vision mezzanine DIP switch (DIP2) setting

DIP switch number	State	Purpose
5	ON	Enable the accelerometer and gyroscope sensor.

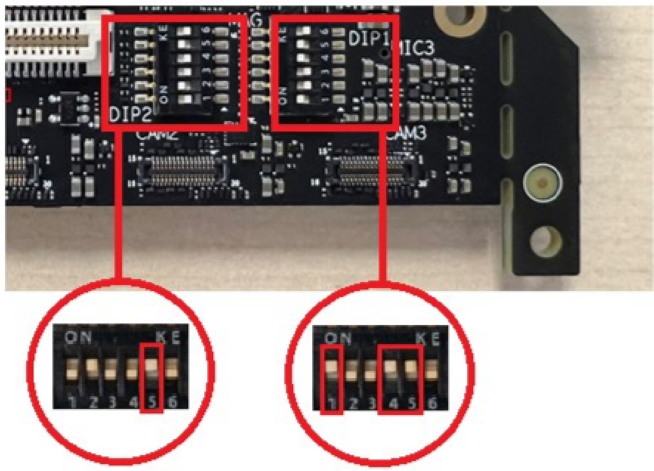

Figure: DIP switch setting to enable all sensors on vision mezzanine board

Software setup: No software change is required to enable the sensors on the Vision kit.

Use an accelerometer and gyroscope on the Vision kit main board (optional)

If you are using the accelerometer and gyroscope sensor on the main board instead of the accelerometer and gyroscope sensor on the vision mezzanine board, then do the following:

Hardware setup: Set up the hardware DIP switches as mentioned in the following table to enable the accelerometer and gyroscope sensor on the main board and disable the accelerometer and gyroscope sensor on the vision mezzanine board.

Table : Main board DIP switch (DIP_SW_0) setting

DIP switch number	State	Purpose
5	OFF	Enable the sensor on the main board.

Table : Vision mezzanine DIP switch (DIP1) setting

DIP switch number	State	Purpose
1	OFF	Disable the accelerometer and gyroscope sensor.

Table : Vision mezzanine DIP switch (DIP2) setting

DIP switch number	State	Purpose
5	OFF	Disable the accelerometer and gyroscope sensor.

Set up software

Note: The `<registry_path>` referred in the following sections should be considered as one of the existing paths on the device: `/etc/sensors/registry/registry/` or `/var/cache/sensors/registry/registry/`.

1. Open the `json.lst` file, remove the `qcm6490_rbx_navmez_icm4x6xx_0.json` entry (if present), and save the `json.lst` file.

   ```
   vi /etc/sensors/config/json.lst
   ```

2. Open the `qcm6490_rbx_icm4x6xx_0.json` file.

   ```
   vi /etc/sensors/config/qcm6490_rbx_icm4x6xx_0.json
   ```

   Add the value 2 into the existing `platform_subtype_id` field as shown here, and save the `qcm6490_rbx_icm4x6xx_0.json` file.

   ```
   "platform_subtype_id": ["5", "2"]
   ```

3. Remove the existing parsed registry files.

   ```
   rm -f <registry_path>/*
   ```

4. Reboot the device.

   ```
   reboot
   ```

5. Check for the registry files that are parsed:

   ```
   ls <registry_path>
   ```

   - The following output is displayed for the main board:

```
qcm6490_rbx_icm4x6xx_0.json.icm4x6xx_0_platform.accel
qcm6490_rbx_icm4x6xx_0.json.icm4x6xx_0_platform.accel.fac_
cal.bias
qcm6490_rbx_icm4x6xx_0.json.icm4x6xx_0_platform.gyro
qcm6490_rbx_icm4x6xx_0.json.icm4x6xx_0_platform.orient
```

- The following output is displayed for the vision mezzanine board:

```
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_platform.accel
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_platform.accel.
fac_cal.bias
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_platform.gyro
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_platform.orient
```

Note: If the accelerometer and gyroscope sensors on the main board are enabled, these sensors on the vision mezzanine board must be enabled. After enabling the sensors, revert or undo all the earlier hardware and software setup steps. After the reboot, on the next bootup verify that multiple files with the `qcm6490_rbx_navmez_icm4x6xx_0` prefix are present under the <registry_path> path (as described in the step 5).

Test sensors on platform

Use the sensor test application to validate the accelerometer, gyroscope, magnetometer, and pressure sensor streaming. By default, the `ssc_drva_test` tool is built under the `/usr/bin/` directory on the device.

- **Test the accelerometer and gyroscope sensor on the Core kit and the Vision kit**

  The following is an example command for the accelerometer sensor:

  ```
  ssc_drva_test -sensor=accel -duration=5 -sample_rate=500
  ```

  The following snippet shows an example command that allows the accelerometer sensor for 5 sec at 500 Hz sample rate in the Streaming mode.

  ```
  root@qcm6490:~# ssc_drva_test -sensor=accel -duration=5 -sample_
  rate=500
  6 ssc_drva_test version 1.27k
  6 ssc_drva_test -sensor=accel -duration=5 -sample_rate=500
  diag: Diag_LSM_Init: invoked for pid: 1141 with init_count: 0
  diag:successfully connected to socket 3
  diag: Diag_LSM_Init: done for pid: 1141 with init_count: 1
  6 event_cb attribute event for da_test
  ```

```
6 event_cb attribute event for da_test
6 using da_test name=da_test, suid = [high addeaddeaddeadde, low
addeaddeaddeadde
6 enter send_memory_log_req cookie: 6
6 exit send_memory_log_req
6 enter da_test runner. -rumifact=1
6 -time_to_first_event=233206
6 -time_to_last_event=-20008
6 -sample_ts=50267544823
6 -total_samples=2528
6 -avg_delta=37875
6 -recvd_phy_config_sample_rate=500
6 -random_seed_used=2926886043
6 -num_request_sent=2
6 -first_sample_timestamp=50171775915
6 received event: PASS
6 enter send_memory_log_req cookie: 6
6 exit send_memory_log_req
6 PASS
```

The following is the output of the test example:

– The `accel` sensor is enabled at 500 Hz. The sensor hardware runs only at certain sample rates described in the sensor hardware data sheet. For example, if the command is modified to request `accel` data at 480 Hz, the `accel` sensor still operates at 500 Hz.

– Total 2528 acceleration samples are received during the test. The following is an example command for the gyroscope sensor:

```
ssc_drva_test -sensor=gyro -duration=5 -sample_rate=500
```

- **Test the magnetometer and pressure sensor on the Vision kit**

  The following is an example command for the magnetometer sensor:

```
ssc_drva_test -sensor=mag -duration=5 -sample_rate=100
```

The following is an example command for the pressure sensor:

```
ssc_drva_test -sensor=pressure -duration=5 -sample_rate=25
```

For troubleshooting common issues, see Troubleshoot sensors.

2 Bring up sensors

2.1 Build and flash an aDSP image

This information is available to licensed users with authorized access. For more information, see Build and flash an aDSP image in Qualcomm Linux Sensors Guide - Addendum.

Note: Qualcomm Linux Sensors Guide - Addendum is accessible only to the licensed users with authorized access.

2.2 Enable sensors on the RB3 Gen 2 Development Kit

For instructions, see Enable sensors on the RB3 Gen 2 Development Kit.

Next steps

- Develop sensors

3 Develop sensors

3.1 QSH client API workflow

The QSH-exposed APIs are called QSH client APIs for applications. The client applications interact with the QSH framework available on the low-power processor. The following figure shows the detailed breakdown and usage of the QSH client APIs:

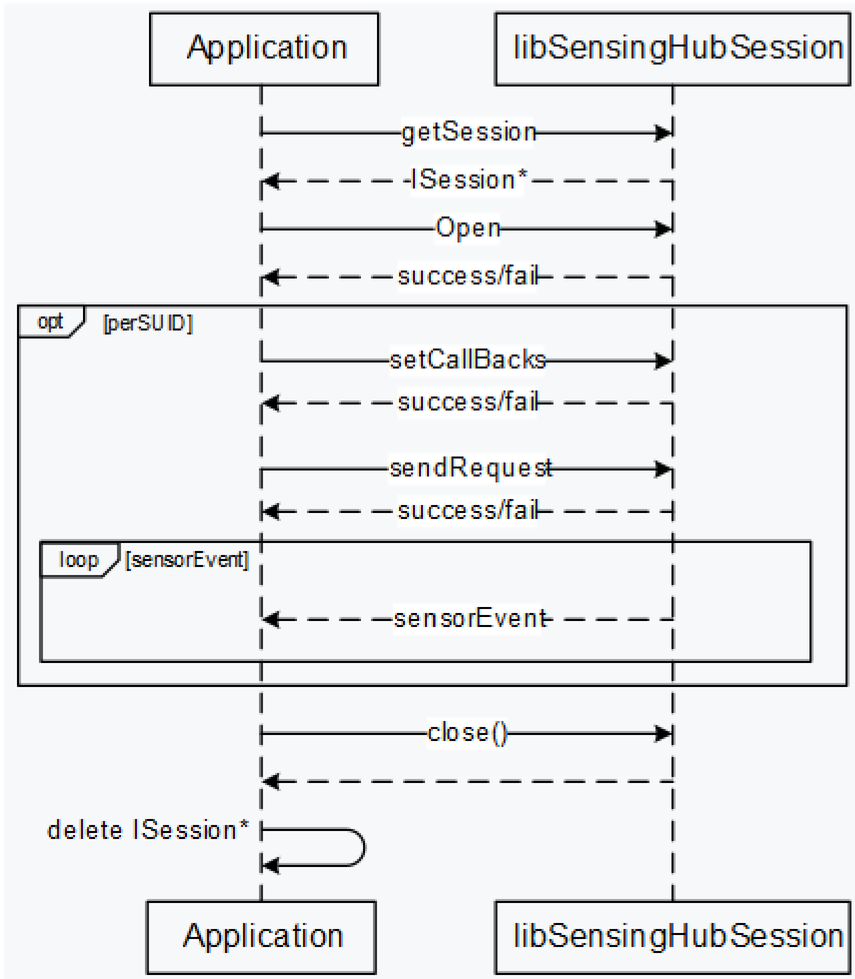

Figure: QSH client API workflow

The following list describes the QSH client APIs:

- **getSession**

  To interact with the QSH, the sensor application client must create an interface session by invoking the `getSession()` API as shown in the figure. Successful creation of the interface session returns an `ISession*` object to the client.

- **Open, Close, SetCallback, and SendRequest**

  The interface created using the `getSession()` API operates on a *once_per_session* basis. This interface allows the application to use the QSH client APIs, such as `open()`, `close()`, `setCallback()`, and `sendRequest()`, until the session is explicitly deleted using the `close()` and `delete()` APIs.

After the interface session establishes, the application or the client code can open a sensor session by calling the `open()` API. Similar to `getSession()`, the `open()` API is also invoked on a *once_per_session* basis. Additionally, the `setCallback()` and `sendRequest()` APIs are invoked once for a specified SUID.

Messages

The sensor client applications sends and receives the following request and response messages:

1. Sends a `sns_client_request_msg` request message through the `sendRequest()` API.
   - The sole field of this message is the payload, which is an opaque byte array. This field is populated with the protocol buffer-encoded `sns_client_request_msg` message.
2. Receives a `sns_client_resp_msg` response message.
   - The client manager sends this response message immediately upon receipt of the request.
   - A minimal amount of processing is performed on the request. The client manager determines if the `sns_client_request_msg` message is appropriately encoded and the destination SUID is available.
3. Receives one or more events of `sns_client_event_msg` message type.
   - The event callback function receives the event messages, as specified in the `setCallback` API.
   - Each message has a SUID.
   - An event message may contain one or more logical sensor samples that are all encoded in the single `sns_client_event_msg` message.

Protocol buffers

The QSH client request and event messages are opaque memory buffers that contain the protocol buffer-encoded messages. Clients can generate these messages in several programming languages and then copy the encoded byte stream into a `sns_client_request_msg` message. Similarly, clients can copy the encoded message from within the `sns_client_report_ind_msg` message and decode it separately.

For more information about protocol buffers, see Protocol-buffers and nanopb.

Client manager

The sensor client manager, housed within the QSH, oversees all communication processes. The client manager responsibilities are:

- **Translating incoming requests**: The client manager receives incoming requests and translates them into a format that the QSH can understand. The translation involves converting the requests into specific request messages.
- **Translating outgoing indications**: When the client manager receives event messages from QSH, the client manager translates the event messages into outgoing indications. The translation ensures that the messages are in a format that's understandable outside the QSH.
- **Guaranteeing batching options**: If a client specifies certain batching options, then the client manager ensures that these options are met. It checks that the data is grouped and sent in the way the client has specified.

  For more information about the client manager, see the `<workspace>/build-qcom-wayland/workspace/sources/sensinghub/sensing-hub/apis/proto/sns_client.proto` file.
- **SUID**: The SUID is an integer that uniquely identifies a single sensor available from the QSH. Clients consider this ID as generated randomly upon boot. Clients don't assume that the ID repeats or derives any information from the number itself.

  For example, if there are two gravity algorithms available from the QSH, then each has its own unique SUID. This unique ID assignment is true for physical sensors too. If the hardware for a physical sensor model is present many times on the device, then each has its own SUID.

  For a complete list of sensors available on the system, the client can send a request to the SUID sensor.
- **Client requests**: All incoming requests to the client manager use `sns_client_request_msg` as the outermost protocol buffer message. This message has the following fields:
  - SUID: Serves as the destination address of the request message. Send all request messages to a valid SUID for processing; otherwise, the client manager rejects them.
  - msg_id: A numeric identifier for the encoded request contained in the `sns_std_request::payload` message.

    For example, a sensor may support an enable streaming request message, and an initiated data flush request. The formats of these two messages are different, and this field indicates the destination sensor on how to interpret the request. Message IDs are always unique among all messages supported by a sensor. However, two different sensors may use a particular ID for different message types.

  For example,

  - **Message types** — there are two types of messages:

    - `SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG`: Denotes standard streaming request from a client to a sensor.
    - `SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_EVENT`: Denotes a standard sensor event from a data source sensor.

  - **Request**—is the information package that the client sends to the sensor. It has all the details that the sensor needs to do its job. The `sns_std_request::payload` field corresponds to the message ID and has the sensor-specific configuration.

    For more information, see the sensor-specific proto.

  - **Resampler configuration**—is used when the client wants to change the rate at which the sensor samples the data. For resampler configuration, use any of the following options:
    - `SNS_RESAMPLER_RATE_FIXED`: The client wants data at a specific rate. For example, if the client wishes to get data at 200 Hz, whereas the physical sensor supports streaming at 240 Hz, then the samples are interpolated to 200 Hz.
    - `SNS_RESAMPLER_RATE_MINIMUM`: The client wants data at a certain least rate. For example, if the client wishes to get data at 200 Hz, whereas the physical sensor supports streaming at 240 Hz, then the samples are sent at 240 Hz.

  - **Threshold configuration**—is used when the client wants the data only when certain conditions are met. There are four types of conditions:
    - `SNS_THRESHOLD_TYPE_RELATIVE_VALUE` provides thresholding as a delta between the current value and the last reported value that exceeds above the configured threshold.
    - `SNS_THRESHOLD_TYPE_RELATIVE_PERCENT` provides thresholding based on the difference between the current value and the last reported value. It's a percentage of the last reported value, with the percentage representing the configured threshold.
    - `SNS_THRESHOLD_TYPE_ABSOLUTE` provides thresholding of the current value against a fixed configured threshold value.
    - `SNS_THRESHOLD_TYPE_ANGLE` provides a thresholding of the angle between the current and the last reported quaternion for the quaternion sensors (in radians).

  - **Suspend configuration**—specifies how the system behaves when the processor suspends.
    - `client_proc_type` is the processor where the client is located. If any client on this processor asks for a flush (a complete send out of data), all clients on that processor get a flush of data.
    - `delivery_type` is about whether to send events while the processor suspends.
      - `SNS_STD_DELIVERY_WAKEUP`: Sends events whenever they become available (at sample rate or batch period). If a `batch_period` larger than the system capacity is requested, all the data is sent upon capacity exhaustion. With this

option, the flush_period is effectively ignored, as unsent batched data that doesn't have the opportunity to accrue in the buffer.

- SNS_STD_DELIVERY_NO_WAKEUP: Sends events only when the client processor isn't suspended; otherwise, batches the data until the flush period. After the target processor exits suspend, all pending events are sent.

- nowakeup_msg_ids is a list of message IDs for which the client processor must not be woken up. The message IDs mentioned aren't wake-up capable. They're only sent if any other wake-up capable events are present.

For more information, see the <workspace>/build-qcom-wayland/workspace/sources/sensinghub/sensing-hub/apis/proto/sns_client.proto file.

- **Batching**: Within the sns_client_request_msg::sns_std_request message, the client can specify how and when it receives the requested data.

  sns_std_request has the following fields:

  – batching::batch_period: An unsigned integer value that denotes the batch time in microseconds. A client can assume that a timer is registered for the given batch_period in microseconds. All events generated since the last timer expiration are saved until the next timer has fired. This period is interpreted as the maximum period specified by the client. Events are delivered to the client at a faster rate (smaller batch period) in some concurrency scenarios. A client may send a flush request at any time to instruct the client manager to send all the batched data. By default, batching is disabled.

  – batching::flush_period: An unsigned integer value whose unit is in microseconds. This field provides a hint to the client manager or physical sensor regarding how much historical data must be batched if the data is not sent to the client. In other words, the client manager may drop the data that is older than the flush_period in microseconds. The effective flush period can be smaller due to the system memory constraints or larger in concurrency cases. This field is optional and if not set, defaults to batch_period, that is, only a single batch of data is maintained. If this field is set, then the flush_period must be greater than or equal to batch_period.

  – batching::flush_only: If set to True, the client manager sends events only upon a client-initiated flush. Otherwise, it continues batching until flush_period is reached (at which time, batching continues, but the oldest data are dropped).

  – batching::max_batch: If set to True, it directs the sensor to operate at maximum batching capacity. If a request has both flush_only = true and max_batch = true, then flush_only takes precedence.

- **Client event**: All outgoing indications from the client manager use sns_client_event_msg as the outermost protocol buffer-encoded message, within the payload field of sns_client_report_ind_msg.

  sns_client_event_msg message has several fields:

- SUID: Associated with the data source event. If a client sends requests to many SUIDs on a single connection, then this message has only the events for one SUID. Events from other SUIDs are delivered in separate indication messages.
- `events::msg_id`: Uniquely identifies the associated event message.
- `events::timestamp`: The timestamp associated with this event in the QTimer clock ticks. For most events, the sensor sets the timestamp, which refers to the time when the physical sample was created in the sensor hardware. For events generated by the framework (such as configuration updates or error events), this timestamp refers to the time at which the event was created.
- `events::payload`: Dynamic length payload, containing the actual data/event from the sensor. Decode this payload separately using the sensor-specific proto buffer.

- **Message payloads**: The sensor-specific request or event is referred in `sns_client_request_msg::sns_std_request::payload` and `sns_client_event_msg::sns_client_event::payload` messages. These message fields contain a protocol buffer-encoded message with fields specific to that message ID or may be empty and have no fields.

  Clients use the `.proto` file associated with the sensor to which they communicate. Each sensor type has a corresponding `.proto` file. For example, `sns_accel.proto` describes how to enable an accelerometer stream. Also, every sensor publishes its list of `.proto` files as a part of its attributes. For more information, see table_sensor_attribute.

  - **Data types**: Each sensor advertises a data type attribute. Each data type is associated with a unique set of `*.proto` files that make up the sensor-specific API for that sensor. All sensors of the same type must support a minimum set of requests and event messages. They may define and use more optional messages that are specific to that sensor implementation. Each sensor publishes a complete list of `.proto` files as part of its attributes.
  - **Standardized messages**: Any client can send a set of standardized Qualcomm-defined messages to any sensor. These messages are defined in the `sns_std.proto` file.
    - `SNS_STD_MSGID_SNS_STD_ATTR_REQ`: Queries a sensor for its list of attributes. It returns an event with an `SNS_STD_MSGID_SNS_STD_ATTR_EVENT` ID that has a list of all the published attributes, see table_sensor_attribute. Clients can also register for notification when a new or updated attribute is published.
      - **`sns_std_attr_req` has `register_updates` field. If set to `True`, the client receives** notifications whenever there is a change in the sensor attributes through `sns_std_attr_event`.
      - **`sns_std_attr_event` has `attributes` field, which lists all the attributes published by a** sensor. It's sent in response to `sns_std_attr_req`, or on an attribute change to a registered sensor.

    - SNS_STD_MSGID_SNS_STD_FLUSH_REQ: Forces all the batch data from this sensor to be immediately sent to the client. This command forces the applicable hardware and software buffers on the system to flush all the data present.

      For example, sending this request to an accelerometer sensor causes the physical FIFO flush, as well as any samples currently held by the client manager. If the flush request is to an algorithm, such as a game rotation vector, which internally uses accelerometer and gyroscope data, then both the accelerometer and gyroscope hardware FIFOs are flushed.

    - SNS_CLIENT_MSGID_SNS_CLIENT_DISABLE_REQ: Disables the active request for this sensor. For example, the client sends a SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG message to the accelerometer sensor to enable streaming. Later, sending a DISABLE_REQ cancels the streaming request, and accelerometer streaming ceases for this client.

    - SNS_STD_MSGID_SNS_STD_FLUSH-EVENT: Response to a flush request. It indicates no further events corresponding to the flush request.

    - SNS_STD_MSGID_SNS_STD_ERROR-EVENT: An error event generated by a sensor/instance or the framework.

      For more information, see the <workspace>/build-qcom-wayland/workspace/sources/sensinghub/sensing-hub/apis/proto/sns_client.proto file.

- **Standardized sensor messages**: The messages described in the client request are applicable to all the sensors. In addition to the messages described in the client request, you can use Qualcomm-recommended standardized messages. These recommendations are optional, and you may instead choose to define your own request and event messages.

    - SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG: Request message that allows streaming for:

        - Any physical sensor, for example, accelerometer, gyroscope, and magnetometer.

        - Some algorithms, for example, rotation vector, gravity, and linear acceleration.

    - SNS_STD_SENSOR_MSGID_SNS_STD_ON_CHANGE_CONFIG: Request message that allows streaming for an on-change type sensor, for example, proximity, ambient_light, and step_detect.

    - SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_PHYSICAL_CONFIG_EVENT: Event sent by all the physical sensors upon processing a client request. It indicates what data stream clients must expect, for example, the rate at which the sensor produces samples.

    - SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_EVENT: Data sample produced by the sensor.

For more information, see the `sns_std_sensor.proto` file.

– **SUID lookup sensor**: Clients may query the SUID lookup sensor for the list of SUIDs associated with a specific data type. The `sns_suid_req` message specifies a data type, and the client receives all matching SUIDs. An empty data type string results in the receipt of the list of all SUIDs on the system.

  For example, if a client specifies *accel* as the data type, then they receive a list of all SUIDs whose sensors provide data of type *accel*. Additionally, if the client wants to receive notifications for a new match, then they can indicate it through the `register_updates` field. The `sns_suid_req` message can be followed up by an attribute request (`sns_std_attr_req`) by the client to decide which of the available *accel* sensors are appropriate for this client.

  The SUID lookup sensor has its own SUID, which is constant and is published in a `sns_suid.proto` file (making it unique among all other sensors).

  The `sns_suid_req` message has the following fields:

Table : SUID request

Field	Mandatory or optional	Data type	Description
`data_type`	Mandatory	String	Data type of the sensor for which SUID is to be queried
`register_updates`	Optional	Boolean	Register for updates to the list of SUIDs advertising the``data_type``field

Field	Mandatory or optional	Data type	Description
`default_only`	Optional	Boolean	Each data type may have one sensor configured to be default through the registry. ◦ If the `default_only` field is set to True and: · A default for the data type is explicitly configured, then only the SUID of the default sensor is sent through the SUID event when available · A default for the data type is not explicitly configured, then the SUID of the first sensor with the matching data type is sent through the SUID event ◦ If the `default_only` field is set to False, then all the sensors with the matching data type are sent as and when they become available.

For more information, see the `sns_suid.proto` file and example code.

- **Sensor attributes**: Every sensor publishes a list of attributes, representing each attribute with a numeric identifier. These attributes provide information regarding the sensor capabilities and the range of values that it accepts as an input.

  The following table lists a few important attributes:

Table : Sensor attribute

Attribute ID	Attribute name	Mandatory?	Data type	Description
0	`SNS_STD_SENSOR_ATTRID_NAME`	Yes	String	Human-readable sensor name.

Attribute ID	Attribute name	Mandatory?	Data type	Description
1	SNS_STD_SENSOR_ATTRID_VENDOR	Yes	String	Human-readable vendor name.
2	SNS_STD_SENSOR_ATTRID_TYPE	Yes	String	The data type used by this sensor, as defined in the sensor proto file.
3	SNS_STD_SENSOR_ATTRID_AVAILABLE	Yes	Boolean	Indicates whether this sensor is available for the clients or not.
4	SNS_STD_SENSOR_ATTRID_VERSION	Yes	Integer	– 64-bit integer value represented as major[31:16].minor[15:8].revision[7:0], denoting the sensor version. – Example in hexadecimal: major: 0x0002 minor: 0x00 revision: 0x36. – DRIVER_VERSION 0x00020036.
5	SNS_STD_SENSOR_ATTRID_API	Yes	String	List of the .proto filenames used by this sensor; more .proto dependencies may be specified as imports within the same proto file used primarily for the test automation.
6	SNS_STD_SENSOR_ATTRID_RATES	No	Float	List of sample rates supported by the sensor.
7	SNS_STD_SENSOR_ATTRID_RESOLUTIONS	No	Float	List of sample resolutions supported by the sensor.

Attribute ID	Attribute name	Mandatory?	Data type	Description
8	SNS_STD_ SENSOR_ ATTRID_ FIFO_SIZE	No	Integer	Supported FIFO depth (in number of samples).
9	SNS_STD_ SENSOR_ ATTRID_ ACTIVE_ CURRENT	No	Integer	Array of active currents (in μA).
10	SNS_STD_ SENSOR_ ATTRID_ SLEEP_ CURRENT	No	Integer	Inactive current (in μA).
11	SNS_STD_ SENSOR_ ATTRID_ RANGES	No	Float	Supported operating ranges by the sensor.
12	SNS_STD_ SENSOR_ ATTRID_ OP_MODES	No	String	An array of strings defines operating modes supported by the sensor. For example, [LPM, HIGH_ PERF, NORMAL, OFF].
13	SNS_STD_ SENSOR_ ATTRID_ DRI	No	Boolean	Denotes whether the sensor supports the DRI or IBI: – True = DRI – False = IBI
14	SNS_STD_ SENSOR_ ATTRID_ STREAM_ SYNC	No	Boolean	Denotes whether a sensor supports synchronized streaming.

Attribute ID	Attribute name	Mandatory?	Data type	Description
15	SNS_STD_ SENSOR_ ATTRID_ EVENT_ SIZE	No	Integer	– The size (in bytes) of the data event (protocol-buffer-encoded) produced by this sensor. – For physical and virtual sensors, this value refers to the size of their sensor sample – Used by the HAL for maximum batching capacity determination.
16	SNS_STD_ SENSOR_ ATTRID_ STREAM_ TYPE	Yes	Integer	Denotes type of streaming supported by the sensor: – 0 = continuous periodic sampling – 1 = on-change – 2 = single output (one-shot)
17	SNS_STD_ SENSOR_ ATTRID_ DYNAMIC	No	Boolean	Specifies if this sensor is dynamic (connected/disconnected at runtime).
18	SNS_STD_ SENSOR_ ATTRID_ HW_ID	No	Integer	Differentiates multiple sensors of the same hardware.
19	SNS_STD_ SENSOR_ ATTRID_ RIGID_ BODY	No	Integer	The rigid body on which the sensor is placed. – 0 = sensor hardware is on the display side – 1 = sensor hardware is on the keyboard side – 2 = sensor hardware is mounted on an external device
21	SNS_STD_ SENSOR_ ATTRID_ PHYSICAL_ SENSOR	No	Boolean	– True, if physical sensor. – False, if virtual sensor.

Attribute ID	Attribute name	Mandatory?	Data type	Description
22	SNS_STD_SENSOR_ATTRID_PHYSICAL_SENSOR_TESTS	No	Integer	List of supported physical sensor tests using enum values in sns_physical_sensor_test_type.
23	SNS_STD_SENSOR_ATTRID_SELECTED_RESOLUTION	No	Float	Measurement resolution for each dynamic range value.
24	SNS_STD_SENSOR_ATTRID_SELECTED_RANGE	No	Float[2]	Dynamic range options supported by the sensor. For the default option, see the requirement specification.
25	SNS_STD_SENSOR_ATTRID_ADDITIONAL_LOW_LATENCY_RATES	No	Float	List of additional sample rates for low-latency operation, in Hz. These sample rates are for dedicated low-latency clients, extending the list of rates published in the``SNS_STD_SENSOR_ATTRID_RATES``attribute. Dedicated internal clients must use these higher data rates, as they may impact system performance.
26	SNS_STD_SENSOR_ATTRID_PASSIVE_REQUEST	No	Boolean	True if the sensor supports passive requests, false otherwise. If a sensor does not support passive requests, then all requests must be treated as active.
29	SNS_STD_SENSOR_ATTRID_TRANSPORT_MTU_SIZE	No	Integers	MTU size for a transport sensor, in bytes.
30	SNS_STD_SENSOR_ATTRID_HLOS_INCOMPATIBLE	No	Boolean	True if the sensor is not compatible with the high-level operating system (HLOS) specification for the supported data type.

Attribute ID	Attribute name	Mandatory?	Data type	Description
31	SNS_STD_SENSOR_ATTRID_SERIAL_NUM	No	String	Sensor serial number.
32	SNS_STD_SENSOR_ATTRID_TECH_USED	No	Integer array	List of technologies used. For more information see, sns_tech``in ``sns_std_type.proto file

For more information, see the sns_std_sensor.proto file and example code.
Proto files are at the /etc/sensors/proto/ location on the device.

3.2 Develop applications using QSH client APIs

The SessionClient sample application shows how to use QSH client API to develop applications. This sample application builds by default in the /usr/bin directory on the device. For complete sample code, see the <workspace>/build-qcom-wayland/workspace/sources/sensinghub/sensing-hub/examples/SessionClient/SessionClient.cpp file.

The following figure shows the call flow for streaming the accelerometer sensor and the usage of the QSH client APIs.

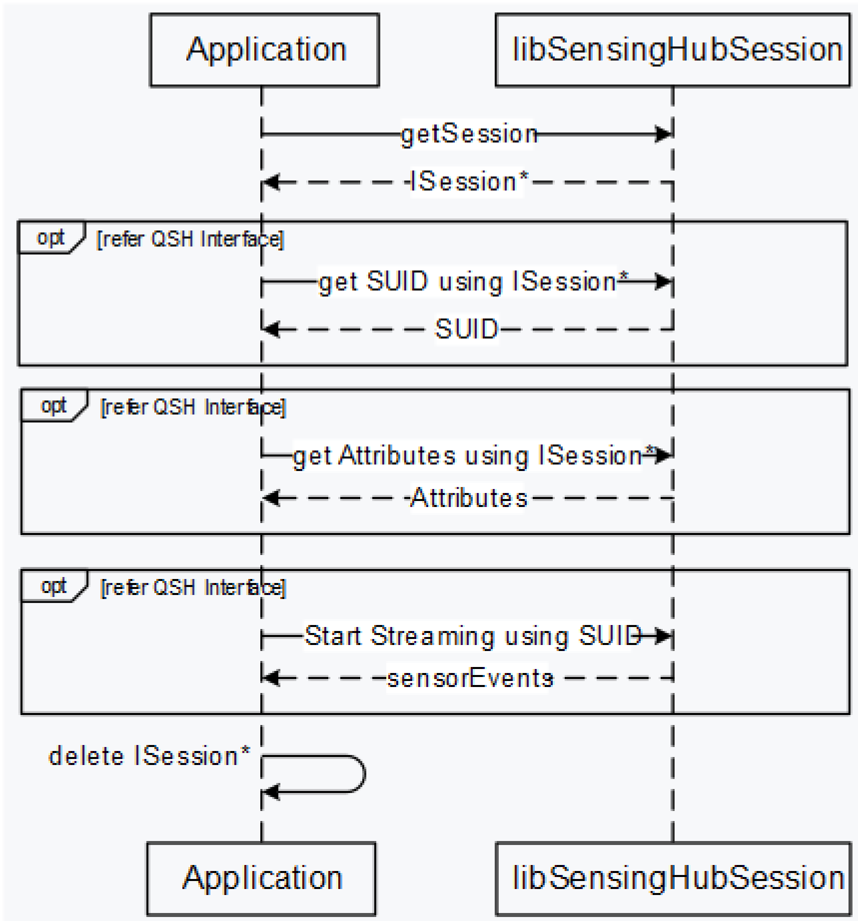

Figure: Call flow to stream a specified sensor

In this example, the client application can use different sensor sessions for the SUID query, attribute query, and streaming activity. It can also use the same session for all the activities; however, synchronization must be handled appropriately. The client application can send various requests to aDSP as follows.

1. **SUID query** retrieves SUIDs for the specified sensor:

   a. Create an interface for SUID by calling the `getSession()` API with the new `sessionFactory()` class. Requesting the SUID is the first and important request to get SUID of the requested data type for any use case.

```
/* Create a new ISession for UID discovery */
 sessionFactory* factory = new sessionFactory();
 if(nullptr == factory){
   printf("failed to create factory instance");
```

```
    return false;
  }
  ISession* suidSession = factory->getSession();
  if(nullptr == suidSession){
    printf("failed to create uid session");
    return false;
  }
```

b. Open a created session interface by calling the `open()` API.

```
/* Open the suidSession */
  int ret = suidSession->open();
  if(-1 == ret){
    printf("failed to open ISession for uid query");
    return false;
  }
```

c. Set callbacks by calling the `setCallBacks()` API and handling the response/event/error for the SUID activity.

```
/* Set callbacks for the session for 'uid' */
  ret = suidSession->setCallBacks(uid, suidResp, nullptr,
suidEvent);
  if(-1 == ret)
     printf("all callbacks are null, no need to register it
");
```

d. Create and send a Pb-encoded request message for SUID of a specified data type by calling the `sendRequest()` API.

```
/*
   * Create SUID request message
   * (Please refer sns_client.proto and sns_suid.proto for
more details)
   * */
  string pb_req_encoded = "";
  sns_suid_req pb_suid_req;
  pb_suid_req.set_data_type(sensorName);
  pb_suid_req.set_register_updates(true);
sns_client_request_msg pb_req_msg;
pb_req_msg.set_msg_id(SNS_SUID_MSGID_SNS_SUID_REQ);
..
string pb_req_msg_encoded;
  pb_req_msg.SerializeToString(&pb_req_msg_encoded);
  /* send proto encoded message to sensing-hub using the
```

```
opened session */
  unique_lock<mutex> respLock(respMutex);
  ret = suidSession->sendRequest(uid, pb_req_msg_encoded);
  if(0 != ret){
    printf("Error in sending uid discovery request");
    return false;
  }
```

e. Close the session by calling the `close()` API and delete it after the SUID events for the requested data type are received.

```
/* Close and delete the session once SUIDs are received */
  suidSession->close();
  delete suidSession;
  delete factory;
```

2. **Attribute request** retrieves attributes for the specified sensor:

a. Create an interface session for an attribute by calling `getSession()` with the new `sessionFactory()` class. Requesting the attributes is important to get the capabilities of the requested data type for any use case.

```
/* Create a new ISession for attribute query */
  sessionFactory* factory = new sessionFactory();
  if(nullptr == factory){
    printf("failed to create factory instance");
    return false;
  }

   ISession* attributeSession = factory->getSession();
  if(nullptr == attributeSession){
    printf("failed to create attribute session");
    return false;
  }
```

b. Open a created session interface by calling the `open()` API.

```
/* open the attributeSession session */
  int ret = attributeSession->open();
  if(-1 == ret){
    printf("failed to open ISession for attribute query");
    return false;
```

c. Set callbacks by calling `setCallBacks()` and handling the response/event/error for the attribute activity.

```
for (const suid& uid : suidList) {
    /* set callbacks for the session for 'uid' */
    int ret = attributeSession->setCallBacks(uid,
attributeResp, nullptr, attributeEvent);
    if(-1 == ret)
         printf("all callbacks are null, no need to register
it");
```

d. Create and send a Pb-encoded configuration request for an attribute of a specified data type by calling the `sendRequest()` API.

```
/* create pb-encoded config request message to be sent for
attribute query */
    sns_client_request_msg pb_req_msg;
    pb_req_msg.set_msg_id(SNS_STD_MSGID_SNS_STD_ATTR_REQ);
    pb_req_msg.mutable_request()->clear_payload();
    pb_req_msg.mutable_suid()->set_suid_high(uid.high);
..
..
/* send proto encoded message to sensing-hub using the opened
session */
    unique_lock<mutex> respLock(respMutex);
    ret = attributeSession->sendRequest(uid, pb_req_msg_
encoded);
```

e. Close the session by calling the `close()` API after the attribute events for the requested data type is received.

```
/* close and delete the session once all attributes are
received */
  attributeSession->close();
  delete attributeSession;
  delete factory;
```

3. **Sensor streaming** streams the sensor and receives the data events:

   a. Create an interface session for streaming the sensor by calling the `getSession()` with the new `sessionFactory()` class. Here, requesting sensor data is the final stage of a requested data type for any use case.

```
sessionFactory()class.

/* create a new ISession for streaming activity */
  sessionFactory* factory = new sessionFactory();
  if(nullptr == factory){
```

```
    printf("failed to create factory instance");
    return false;
  }
  ISession* streamingSession = factory->getSession();
  if(nullptr == streamingSession){
    printf("failed to create streaming session");
    return false;
  }
```

b. Open a created session interface by calling the `open()` API.

```
/* open the streamingSession session */
int ret = streamingSession->open();
if(-1 == ret){
  printf("failed to open ISession for attribute query");
  return false;
```

c. Set callbacks by calling `setCallBacks()` and handling the response/event/error for streaming the activity.

```
for (const suid& uid : suidList){
    /* set callbacks for the session for 'uid' */
    int ret = streamingSession->setCallBacks(uid, dataResp,
dataError, dataEvent);
    if(-1 == ret)
      printf("all callbacks are null, no need to register it
");
```

d. Create and send a Pb-encoded configuration request for streaming the sensor of a specified data type by calling the `sendRequest()` API, which eventually allows the requested sensor.

```
/* create pb-encoded config request message to be sent for
streaming request */
    string pb_req_encoded = "";
    sns_std_sensor_config pb_stream_cfg;
    pb_stream_cfg.set_sample_rate(sampleRate);
    pb_stream_cfg.SerializeToString(&pb_req_encoded);
    sns_client_request_msg pb_req_msg;
    pb_req_msg.mutable_request()->mutable_batching()->set_
batch_period(batchPeriod);
    pb_req_msg.set_msg_id(SNS_STD_SENSOR_MSGID_SNS_STD_
SENSOR_CONFIG);
..
..
```

```
/* send proto encoded message to sensing-hub using the opened
session */
    unique_lock<mutex> respLock(respMutex);
    ret = streamingSession->sendRequest(uid, pb_req_msg_
encoded);
```

e. Handle samples in the event callbacks and wait for the specified duration of the test.

```
void handle_event_cb(const uint8_t *data, size_t size,
uint64_t time_stamp){
if(true == deletion_started){
printf("\nEvent coming when deletion of qmi connection
started");
return;
}
sns_client_event_msg pb_event_msg;
/* Parse the pb encoded event */
pb_event_msg.ParseFromArray(data, size);
/* Iterate over all events in the message */
for (int i = 0; i < pb_event_msg.events_size(); i++) {
auto& pb_event = pb_event_msg.events(i);
```

4. **Stop sensor streaming** client stops streaming by sending a disable request:

   a. Call the `sendRequest()` API.

```
pb_req_msg.set_msg_id(SNS_CLIENT_MSGID_SNS_CLIENT_DISABLE_
REQ);
    pb_req_msg.mutable_suid()->set_suid_high(uid.high);
    pb_req_msg.mutable_suid()->set_suid_low(uid.low);
..
..
/* send disable request to sensing-hub */
    int ret = streamingSession->sendRequest(uid, pb_req_msg_
encoded);
```

   b. Close the session by calling the `close()` API after the streaming events for the requested data type is received.

```
/* close and delete the streamingSession */
  streamingSession->close();
  delete streamingSession;
  delete factory;
```

The following snippet shows the `SessionClient` sample application output. It allows an accelerometer sensor with 10 Hz sample rate and a 2-second batch period for 10 sec and prints the received sensor events.

```
root@qcm6490:~# SessionClient
Streaming configuration is as follows :
        Sensor name : accel     Sample rate : 10 Hz     Batch period
: 2 sec    Test duration : 10 sec


SUID discovery response received.
Received SUIDs for accel, number of SUIDs received = 1

SUID received - suid_low=6360260105974108950 suid_
high=7037810611998542250
Sensor suid list created

requesting attributes for - suid_low=6360260105974108950 suid_
high=7037810611998542250

Attribute query response received.
Attributes for - suid_low=6360260105974108950 suid_
high=7037810611998542250 are:
attribute count 0        and values are: attr_id: 16     sint: 0
attribute count 1        and values are: attr_id: 9      sint:
50sint: 240sint: 240
attribute count 2        and values are: attr_id: 12     std: LPM
std: NORMAL std: HIGH_PERF
attribute count 3        and values are: attr_id: 5      std: sns_
accel.proto
attribute count 4        and values are: attr_id: 0      std:
icm4x6xx
attribute count 5        and values are: attr_id: 1      std: TDK-
Invensense
attribute count 6        and values are: attr_id: 26     boolean 1
attribute count 7        and values are: attr_id: 17     boolean 0
attribute count 8        and values are: attr_id: 10     sint: 6
attribute count 9        and values are: attr_id: 15     sint: 16
attribute count 10       and values are: attr_id: 21     boolean 1
attribute count 11       and values are: attr_id: 22     sint: 3sint:
2sint: 1
attribute count 12       and values are: attr_id: 2      std: accel
attribute count 13       and values are: attr_id: 4      sint: 82179
attribute count 14       and values are: attr_id: 13     boolean 1
attribute count 15       and values are: attr_id: 14     boolean 0
attribute count 16       and values are: attr_id: 18     sint: 0
```

```
attribute count 17       and values are: attr_id: 20     flt: 0.
000000  flt: 0.000000   flt: 0.000000   flt: 0.000000   flt: 0.000000
  flt: 0.000000flt: 0.000000    flt: 0.000000   flt: 0.000000   flt:
0.000000   flt: 0.000000   flt: 0.000000
attribute count 18       and values are: attr_id: 19     sint: 0
attribute count 19       and values are: attr_id: 11
attribute count 20       and values are: attr_id: 7      flt: 0.
000019  flt: 0.000037   flt: 0.000075   flt: 0.000150   flt: 0.000299
attribute count 21       and values are: attr_id: 24
attribute count 22       and values are: attr_id: 23     flt: 0.
000299
attribute count 23       and values are: attr_id: 6      flt: 12.
500000 flt: 25.000000  flt: 50.000000  flt: 100.000000 flt: 200.
000000 flt: 500.000000
attribute count 24       and values are: attr_id: 25     flt: 1000.
000000       flt: 2000.000000
attribute count 25       and values are: attr_id: 8      sint: 80
attribute count 26       and values are: attr_id: 3      boolean 1

Attributes for all SUIDs received


Streaming started
sending request for - suid_low=6360260105974108950 suid_
high=7037810611998542250
Data request response received.
Received re-configuration event
Cal event packet received
Received Samples:       [0.347159],     [-0.181959],    [9.450213],
Received Samples:       [0.102951],     [-0.183156],    [9.545981],
Received Samples:       [0.096965],     [-0.189142],    [9.550770],
Received Samples:       [0.092177],     [-0.192733],    [9.541193],
Received Samples:       [0.090980],     [-0.183156],    [9.555558],
Received Samples:       [0.093374],     [-0.185551],    [9.555558],
Received Samples:       [0.108936],     [-0.184354],    [9.541193],
Received Samples:       [0.098162],     [-0.185551],    [9.565135],
Received Samples:       [0.095768],     [-0.185551],    [9.550770],
Received Samples:       [0.100556],     [-0.193930],    [9.550770],
Received Samples:       [0.092177],     [-0.186748],    [9.550770],
Received Samples:       [0.094571],     [-0.199916],    [9.541193],
Received Samples:       [0.105345],     [-0.199916],    [9.550770],
Received Samples:       [0.098162],     [-0.185551],    [9.550770],
```

For troubleshooting common issues, see Troubleshoot sensors.

For more information, see QSH direct channel API workflow in Qualcomm Linux Sensors Guide - Addendum. This information is available to licensed users with authorized access.

3.3 Configure sensors

This information is available to licensed users with authorized access. For more information, see Configure sensors in Qualcomm Linux Sensors Guide - Addendum.

3.4 Develop and integrate sensor drivers

This information is available to licensed users with authorized access. For more information, see Develop and integrate sensor drivers in Qualcomm Linux Sensors Guide - Addendum.

3.5 Develop and integrate custom sensor algorithms

This information is available to licensed users with authorized access. For more information, see Develop and integrate custom sensor algorithms in Qualcomm Linux Sensors Guide - Addendum.

3.6 Calibrate sensors

Sensors that are sensitive to changes in the operating environment produce inappropriate output values due to factors, such as temperature variations, mechanical wear, or shift in the operating range.

Sensor calibration allows to do the following:

- Adjust or fine-tune the sensor hardware to ensure accuracy and minimize the errors.
- Compare the expected output based on the theoretical models or standards with the actual measured output from the sensor.
- Enhance performance, accuracy, and reliability, which ensures that the sensor provides reliable data.

On successful calibration and associated registry update (as described in the following sections), the value of ver registry item increments. For example, if the bias offset in registry item has ver value of 0, then on a successful calibration and registry update, the registry item is updated with the new bias offset and the ver value increments to 1.

Note: The `<registry_path>` referred in the following sections should be considered as one of the existing paths on the device: `/etc/sensors/registry/registry/` or

/var/cache/sensors/registry/registry/.

Perform factory sensor calibration

The factory sensor calibration is a process to adjust or fine-tune the sensors during the manufacturing phase.

Accelerometer factory calibration

The following table lists the accelerometer calibration procedure:

Table : Accelerometer calibration

Command	Procedure	Bias values
`ssc_drva_test -sensor=accel -factory_test=2 -duration=10` **Note:** A test result (PASS or FAIL) indicates only the test execution status whether the test is completed or not.	Run the command while keeping the device stationary on a flat surface.	The following is the example of the QCS6490 device (ICM42688 sensor): The <registry_path>/qcs6490_rbx_icm4x6xx_0.json.icm4x6xx_0_platform.accel.fac_cal.bias file is created/updated automatically, and stores the bias offset after the calibration.

Magnetometer factory calibration

The following table lists the magnetometer calibration procedure:

Table : Magnetometer calibration

Command	Procedure	Bias values
NA	1 Get */calculate corr_matrix* and *bias scale* values (if applicable) for your device with the help of the sensor vendor. 2 Set the values into the corr_matrix section in the platform-specific magnetometer JSON file of the sensor.	The following is an example of the QCS6490 device (AK09915 sensor): The <registry_path>/qcs6490_rbx_navmez_ak991x_0.json.ak0991x_0_platform.mag.fac_cal.corr_mat file is created/updated automatically, and stores the bias offset after the calibration.

Proximity factory calibration

The following table lists the proximity calibration procedure:

Table : Proximity calibration

Command	Procedure	Bias values
`ssc_drva_test –sensor=prox –factory_test=2 –duration=10`	1 Keep an obstacle/object at a required distance (for example, 5cm or as mentioned in the specification) from the proximity sensor and run the command. 2 Verify that the test returns a PASS result. 3 This procedure uses 5cm as the threshold and the distance within the threshold is considered as near and the distance more than the threshold is considered as far.	The following is an example of the TMD3702 sensor: The <registry_path>/tmd3702_platform.prox.fac_cal file is created/updated automatically, and stores the bias offset after the calibration.

Ambient light factory calibration

The following table lists the ambient light calibration procedure:

Table : Ambient light calibration

Command	Procedure	Bias values
`ssc_drva_test -sensor=als -factory_test=2 -duration=10`	Confirm the test procedure specified by the sensor vendor and run the command.	The following is an example of the TMD3702 sensor: The `<registry_path>/tmd3702_platform.als.fac_cal` file is created/updated automatically, and stores the bias offset after the calibration.

Perform runtime sensor calibration

Runtime sensor calibration adjusts or fine-tunes the sensors during their operational phase, rather than during the manufacturing phase. It occurs dynamically while the sensor actively collects the data in the real-world scenarios.

Gyroscope runtime calibration

The gyroscope calibration sensor `gyro_cal` is used for gyroscope runtime calibration. The following table lists the gyroscope calibration procedure:

Table : Gyroscope calibration

Command	Procedure	Bias values
Use `see_workhorse` to enable a gyroscope calibration sensor (`gyro_cal`). `see_workhorse -sensor=gyro_cal -on_change=1 -duration=120 -display_events=1`	• While keeping the device stationary, request the gyroscope calibration sensor `gyro_cal` data for more than 120 sec and then stop. • After the command is executed, check the status field in the received event for the calibration accuracy status. **Note:** The gyroscope sensor always produces noncalibrated values. Hence,	The `<registry_path>/sns_gyro_cal_persist_sX.bias` file (where, X = sensor index) is created/updated automatically, and stores the bias offset after the calibration: **Note:** Gyroscope calibration runs approximately for every 60sec. The gyroscope bias is stored into the registry only after the existing streaming clients for the gyroscope are disabled.

Command	Procedure	Bias values

Magnetometer runtime calibration

The magnetometer calibration sensor `mag_cal` is used for the magnetometer runtime calibration. The following table lists the magnetometer calibration procedure:

Table : Magnetometer calibration

Command	Procedure	Bias values
Use `see_workhorse` to enable the magnetometer calibration sensor `mag_cal`. `see_workhorse -sensor=mag_cal -on_change=1 -duration=500 -display_events=1`	• Bring the device to an open area where there is no magnetic field interference. • Stream the magnetometer data and move the device in motion **8** for more than 60sec. Moving the device in motion **8** is mandatory. • After the command is executed, check the status field in the received event for the calibration accuracy status.	The `<registry_path>/sns_mag_cal_persist_sXmY.bias` file (where, X = sensor index and Y = device mode index) is created/updated automatically, and stores the bias offset after the calibration.

4 Test and troubleshoot

4.1 Tools

The following tools are supported for functional testing and build environment setup.

Sensor functional test tools

The sensor info test, driver acceptance test, and sensor workhorse tool facilitates comprehensive testing of the sensor functionalities.

- The sensor info test provides detailed information about the sensor.
- The driver acceptance test ensures that the sensor drivers are functioning correctly and are compatible with the QSH.
- The sensor workhorse is a tool to stress-test the sensors under various conditions.

The test tools ensure the accuracy, reliability, and optimal performance of both the hardware-based and software-based sensors in your applications. The following sections provide information on the test tools.

Sensor info test (ssc_sensor_info) to list QSH supported sensors and their attributes

Sensor info test definition

The sensor info test `ssc_sensor_info` is an application within the QSH test suite. It lists the QSH supported sensors and their attributes. You can query the attributes of a specified data type.

```
ssc_sensor_info [-sensor=<sensor_type>] [-delay=<time_in_seconds>][-
duration=<time_in_seconds>][-default_only=<1|0>][-log=<1|0>][-help]
```

Table : Parameters of ssc_sensor_info

Flags	Type	Value	Units	Notes
Sensor	`string`	Any valid sensor type, such as *accel* and *gyro*.	–	Queries the attribute information for a specified sensor type.
Log	`int`	0 | 1	–	Enables or disables diagnostic (API) logs.
help(h)	`int`	–	–	Displays command usage help.
Duration	`int`	Positive values	seconds	Specifies the wait time for the sensor attribute updates.
Delay	`int`	Positive values	seconds	Specifies the time delay before sending the sensor requests.
default_ only	`int`	0 | 1	–	If the `default_only` flag is set to False, UIDs of all the available sensors that support the specified data type are sent. If the `default_only` flag is set to True, only the UID of the default sensor availability is sent.

Sensor info test examples

- Query the sensor attributes and generate diagnostic logs.

```
ssc_sensor_info
```

```
root@qcm6490:~# ssc_sensor_info
ssc_sensor_info v1.86

SUID              = 0xebd5604d09d379bca54dcf30ec041e0f
NAME              = ak0991x
VENDOR            = akm
TYPE              = mag
AVAILABLE         = true
VERSION           = 2.62.47
API               = sns_mag.proto
RATES             = [1.000000, 10.000000, 20.000000, 50.000000,
100.000000]
RESOLUTIONS       = 0.150000
RANGES            = [-4912.000000,4912.000000]
DRI               = false
FIFO_SIZE         = 0
STREAM_TYPE       = streaming
STREAM_SYNC       = false
```

```
DYNAMIC        = false
EVENT_SIZE     = 16
OP_MODES       = [LOW_POWER, LOW_NOISE]
ACTIVE_CURRENT = 900
SLEEP_CURRENT  = 3
HW_ID          = 0
RIGID_BODY     = display
PHYSICAL_SENSOR       = true
PHYSICAL_SENSOR_TESTS = [3, 1]
SELECTED_RESOLUTION   = 0.150000
SELECTED_RANGE        = [-4912.000000,4912.000000]

SUID           = 0xaddeaddeaddeaddeaddeaddeaddeadde
NAME           = da_test
VENDOR         = QTI
TYPE           = da_test
AVAILABLE      = true
VERSION        = 0.51.0
API            = sns_da_test.proto
STREAM_TYPE    = on_change

SUID           = 0x69def905fea2fbac6a43ca273221a2eb
NAME           = aont
VENDOR         = QTI
TYPE           = aont
AVAILABLE      = true
VERSION        = 0.0.1
API            = sns_aont.proto
STREAM_TYPE    = on_change

SUID           = 0xabababababababababababababababab
NAME           = suid
VENDOR         = QTI
TYPE           = suid
AVAILABLE      = true
VERSION        = 0.0.1
API            = sns_suid.proto
STREAM_TYPE    = single_output

SUID           = 0xe12754a7007f27595e2541b4701e2275
NAME           = registry
VENDOR         = QTI
TYPE           = registry
AVAILABLE      = true
```

```
VERSION         = 0.0.1
API             = sns_registry.proto
STREAM_TYPE     = single_output

SUID            = 0xadfeadfeadfeadfeadfeadfeadfeadfe
NAME            = da_test_big_image
VENDOR          = QTI
TYPE            = da_test
AVAILABLE       = true
VERSION         = 0.49.0
API             = sns_da_test.proto
STREAM_TYPE     = on_change

SUID            = 0x61ab5376b4a5c9aa58442ede47acd316
NAME            = icm4x6xx
VENDOR          = TDK-Invensense
TYPE            = accel
AVAILABLE       = true
VERSION         = 1.65.2
API             = sns_accel.proto
RATES           = [12.500000, 25.000000, 50.000000, 100.000000,
200.000000, 500.000000]
RESOLUTIONS     = [0.000019, 0.000037, 0.000075, 0.000150, 0.
000299]
RANGES          = [[-9.806650,9.806650], [-19.613300,19.613300],
[-39.226601,39.226601], [-78.453201,78.453201], [-156.906403,
156.906403]]
DRI             = true
FIFO_SIZE       = 200
STREAM_TYPE     = streaming
STREAM_SYNC     = false
DYNAMIC         = false
EVENT_SIZE      = 16
OP_MODES        = [LPM, NORMAL, HIGH_PERF]
ACTIVE_CURRENT = [50, 240, 240]
SLEEP_CURRENT   = 6
HW_ID           = 0
RIGID_BODY      = display
PHYSICAL_SENSOR       = true
PHYSICAL_SENSOR_TESTS = [3, 2, 1]
SELECTED_RESOLUTION   = 0.000299
SELECTED_RANGE        = [-156.906403,156.906403]
LOW_LATENCY_RATES     = 1000.000000
PASSIVE_REQUEST       = true
```

- Query all *accel* sensor attributes.

```
ssc_sensor_info -sensor=accel
```

```
root@qcm6490:~# ssc_sensor_info -sensor=accel
ssc_sensor_info v1.86

SUID              = 0x61ab5376b4a5c9aa58442ede47acd316
NAME              = icm4x6xx
VENDOR            = TDK-Invensense
TYPE              = accel
AVAILABLE         = true
VERSION           = 1.65.2
API               = sns_accel.proto
RATES             = [12.500000, 25.000000, 50.000000, 100.000000,
200.000000, 500.000000]
RESOLUTIONS       = [0.000019, 0.000037, 0.000075, 0.000150, 0.
000299]
RANGES            = [[-9.806650,9.806650], [-19.613300,19.613300],
[-39.226601,39.226601], [-78.453201,78.453201], [-156.906403,
156.906403]]
DRI               = true
FIFO_SIZE         = 80
STREAM_TYPE       = streaming
STREAM_SYNC       = false
DYNAMIC           = false
EVENT_SIZE        = 16
OP_MODES          = [LPM, NORMAL, HIGH_PERF]
ACTIVE_CURRENT    = [50, 240, 240]
SLEEP_CURRENT     = 6
HW_ID             = 0
RIGID_BODY        = display
PHYSICAL_SENSOR       = true
PHYSICAL_SENSOR_TESTS = [3, 2, 1]
SELECTED_RESOLUTION   = 0.000299
SELECTED_RANGE        = [-156.906403,156.906403]
LOW_LATENCY_RATES     = [1000.000000, 2000.000000]
PASSIVE_REQUEST       = true
```

Output

The test application generates the following output:

- Standard output on console

- Result file

Driver acceptance test (ssc_drva_test) to validate sensor drivers

Driver acceptance test definition

The `ssc_drva_test` driver acceptance test tool does the following:

- Perform sensor driver validation and operate at the QSH sensor API layer.
- Execute a range of sensor use cases.
- Accept the parameters directly from the command line and eliminate the need for compile-time options.

This approach makes it a convenient and efficient tool to perform basic driver-level tests or validations.

```
ssc_drva_test [-sensor=<sensor_type>] [-duration=<time_in_seconds>]
[-sample_rate=<value_in_Hz>] [-batch_period=<value_in_seconds>] [-
iterations=<value>] [-num_samples=<value>] [-factory_test=<value>]
```

Table : Parameters of ssc_drva_test

Flag	Type	Value	Unit	Notes
`sensor`	`string`	• `accel` • `gyro` • `sensor_temperature` • `pressure` • `mag` • `humidity` • `ambient_temperature` • `ultra_violet` • `proximity` • `ambient_light` • `rgb` • `hall` • Any custom sensor added	–	Mandatory argument: limited to the available sensor types.
`duration`	`float`	Positive values only	Sec	Mandatory argument: sensor test duration in seconds.

Flag	Type	Value	Unit	Notes
`sample_rate`	`float`	Positive floating point number values: • -1: Maximum sampling rate • -2: Minimum sampling rate	Hz	Mandatory for streaming sensors, optional for on-change sensors.
`batch_period`	`float`	Positive floating point numbers	Sec	This period is the same as the report period and indicates how long to buffer the samples and report outside of the low-power processor.
`iterations`	`int`	Positive values only	NA	Provides the number of times the test must be repeated.

Flag	Type	Value	Unit	Notes
num_samples	int	Positive values only	NA	Indicates the minimum number of samples intended to be collected. If a num_samples parameter is specified and the test does not collect enough samples during the test, the test sensor generates FAIL. The num_samples parameter forces the test to run for a maximum duration between a specified duration or duration calculated by the following: • num_samples* expected sample rate, where the expected sample rate is the rate at which the sensor is expected to serve.

Flag	Type	Value	Unit	Notes
factory_test	int	• 0 (SNS_PHYSICAL_SENSOR_TEST_TYPE_SW) • 1 (SNS_PHYSICAL_SENSOR_TEST_TYPE_HW) • 2 (SNS_PHYSICAL_SENSOR_TEST_TYPE_FACTORY) • 3 (SNS_PHYSICAL_SENSOR_TEST_TYPE_COM)	NA	Selects the type of factory test (from the value column) that you want to run.

Driver acceptance test examples

- Stream a single sensor at a selected sampling frequency for a known duration:

```
ssc_drva_test -sensor=accel -duration=5 -sample_rate=500
```

```
root@qcm6490:~# ssc_drva_test -sensor=accel -duration=5 -sample_
rate=500
6 ssc_drva_test version 1.27k
6 ssc_drva_test -sensor=accel -duration=5 -sample_rate=500
diag: Diag_LSM_Init: invoked for pid: 1141 with init_count: 0
diag:successfully connected to socket 3
diag: Diag_LSM_Init: done for pid: 1141 with init_count: 1
6 event_cb attribute event for da_test
6 event_cb attribute event for da_test
6 using da_test name=da_test, suid = [high addeaddeaddeadde, low
```

```
addeaddeaddeadde
6 enter send_memory_log_req cookie: 6
6 exit send_memory_log_req
6 enter da_test runner. -rumifact=1
6 -time_to_first_event=233206
6 -time_to_last_event=-20008
6 -sample_ts=50267544823
6 -total_samples=2528
6 -avg_delta=37875
6 -recvd_phy_config_sample_rate=500
6 -random_seed_used=2926886043
6 -num_request_sent=2
6 -first_sample_timestamp=50171775915
6 received event: PASS
6 enter send_memory_log_req cookie: 6
6 exit send_memory_log_req
6 PASS
```

- Batch a single sensor at a selected sampling frequency and report period for a known duration:

```
ssc_drva_test -sensor=accel -duration=30.0 -sample_rate=100 -
batch_period=2.0
```

```
root@qcm6490:~# ssc_drva_test -sensor=accel -duration=30.0 -
sample_rate=100 -batch_period=2.0
3 ssc_drva_test version 1.27k
3 ssc_drva_test -sensor=accel -duration=30.0 -sample_rate=100 -
batch_period=2.0
diag: Diag_LSM_Init: invoked for pid: 1413 with init_count: 0
diag:successfully connected to socket 3
diag: Diag_LSM_Init: done for pid: 1413 with init_count: 1
3 event_cb attribute event for da_test
3 event_cb attribute event for da_test
3 using da_test name=da_test, suid = [high addeaddeaddeadde, low
addeaddeaddeadde
3 enter send_memory_log_req cookie: 3
3 exit send_memory_log_req
3 enter da_test runner. -rumifact=1
3 -time_to_first_event=15321140
3 -time_to_last_event=-13606286
3 -sample_ts=8591389823
3 -total_samples=2960
3 -avg_delta=184826
```

```
3 -recvd_phy_config_sample_rate=100.000000
3 -random_seed_used=3720387971
3 -num_request_sent=2
3 -first_sample_timestamp=8030694123
3 received event: PASS
3 enter send_memory_log_req cookie: 3
3 exit send_memory_log_req
3 PASS
```

- Self-test for accelerometer (hardware self-test):

```
ssc_drva_test -sensor=accel -factory_test=1 -duration=10
```

```
root@qcm6490:~# ssc_drva_test -sensor=accel -factory_test=1 -
duration=10
4 ssc_drva_test version 1.27k
4 ssc_drva_test -sensor=accel -factory_test=1 -duration=10
diag: Diag_LSM_Init: invoked for pid: 1439 with init_count: 0
diag:successfully connected to socket 3
diag: Diag_LSM_Init: done for pid: 1439 with init_count: 1
4 event_cb attribute event for da_test
4 event_cb attribute event for da_test
4 using da_test name=da_test, suid = [high addeaddeaddeadde, low
addeaddeaddeadde
4 enter send_memory_log_req cookie: 4
4 exit send_memory_log_req
4 enter da_test runner. -rumifact=1
4 -time_to_first_event=0
4 -time_to_last_event=-2032137957
4 -sample_ts=10622073335
4 -total_samples=0
4 -avg_delta=0
4 -recvd_phy_config_sample_rate=0.000000
4 -random_seed_used=1840104635
4 -num_request_sent=2
4 -first_sample_timestamp=0
4 received event: PASS
4 enter send_memory_log_req cookie: 4
4 exit send_memory_log_req
4 PASS
```

Output

On the console command line, this test only outputs as pass or fail, which indicates only the test execution status (whether the test is complete).

Sensor workhorse (see_workhorse) to perform stress test of sensors

Sensor workhorse definition

The `see_workhorse` sensor workhorse tool does the following: - Operates specific sensors based on the command-line arguments. - Streamlines sensor testing and data collection in various configurations.

```
see_workhorse [-sensor=<sensor_type>][-sample_rate=<min | max |
number>] [-batch_period=<seconds>] [-calibrated=<0 | 1>] [-wakeup=<0
| 1>]
```

Table : Parameters of see_workhorse

Flags	Type	Value range	Units	Notes
sensor	string	• accel • gyro • sensor_temperature • pressure • mag • humidity • ambient_temperature • ultra_violet • proximity • ambient_light • rgb • hall • Any custom sensor added	NA	Mandatory argument: Limited to the available sensor types.

Flags	Type	Value range	Units	Notes
on_change	int	0 | 1	NA	• 0 selects using SNS_STD_SENSOR_STREAM_TYPE_STREAMING. • 1 selects using SNS_STD_SENSOR_STREAM_TYPE_ON_CHANGE.
sample_rate	float	Positive floating point number values: • -1: Maximum sampling rate • -2: Minimum sampling rate	Hz	Mandatory for streaming sensors, optional for on-change sensors.
batch_period	float	Positive floating point numbers	Seconds	Same as the batch period or report period.

Flags	Type	Value range	Units	Notes
calibrated	int	0 | 1	NA	• 0: nop (default). • 1: if the sensor_type is gyro or mag, then also activate gyro_cal or / mag_cal, respectively.
wakeup	int	0 | 1	NA	• 0: sets suspend_config wake-up to SEE_CLIENT_DELIVERY_NO_WAKEUP. • 1: sets suspend_config wake-up to SEE_CLIENT_DELIVERY_WAKEUP (default).
display_events	int	0 | 1	NA	Display sensor events in JSON format, using the event callbacks.

Sensor workhorse examples

For instance, use the following command to stream the accelerometer data at the maximum rate for

30 sec:

```
see_workhorse -sensor=accel -sample_rate=max -duration=30 -display_
events=1
```

```
root@qcm6490:~# see_workhorse -sensor=accel -sample_rate=max -
duration=30 -display_events=1
see_workhorse version 2.04
20:22:15.406 see_workhorse -sensor=accel -sample_rate=max -
duration=30 -display_events=1
begin usta_get_sensor_list
```

This command instructs `see_workhorse` to run the accelerometer `accel` at a maximum sample rate for a duration of 30 sec.

```
20:22:15.731 stream_sensor( accel)
20:22:15.731 + sample_rate: 500.000000 hz
20:22:15.838 config_stream_sensor() complete rc 0
20:22:15.838 sleep(30) seconds

"sns_client_event_msg" : {
 "suid" : {
  "suid_low" : "0x58442ede47acd316",
  "suid_high" : "0x61ab5376b4a5c9aa"
 },
 "events" : [
  {
   "msg_id" : 768,
   "timestamp" : 78497742089,
   "payload" : {
    "sample_rate" : 500.000000,
    "water_mark" : 1,
```

Miscellaneous tools

Use the following tools for the build environment and functional diagnosis:

- **Qualcomm® Hexagon™ DSP toolchain**

  Hexagon 8.4.07 toolchain

- **QXDM Professional tool**

  The Qualcomm® eXtensible diagnostic monitor (QXDM) Professional™ tool is a diagnostic client.

  – The QSH framework can call the macros and APIs directly to send the debug information to print strings and log packets.

  – The QXDM logs are primarily used for the aDSP side debugging. For more information, see Troubleshoot sensors in Qualcomm Linux Sensors Guide - Addendum.

  > **Note:** Qualcomm Linux Sensors Guide - Addendum is accessible only to the licensed users.

  – The QXDM Professional Tool requires a USB connection. The application processors have direct connectivity to a USB port, whereas the aDSP doesn't have such connectivity.

4.2 Verify

You can determine whether the configured QSH functionalities are operating properly or require attention for any discrepancies using the tools available for verifying these functionalities. You can use the test tool or the QXDM Professional to examine the configured settings and take the necessary action.

The `ssc_sensor_info` test tool verifies the QSH functionality and lists the supported sensors with their capabilities.

```
ssc_sensor_info
```

The following shell output lists all the supported sensors and their attributes:

```
root@qcm6490:~# ssc_sensor_info
ssc_sensor_info v1.86

SUID              = 0xebd5604d09d379bca54dcf30ec041e0f
NAME              = ak0991x
VENDOR            = akm
TYPE              = mag
```

```
AVAILABLE      = true
VERSION        = 2.62.47
API            = sns_mag.proto
RATES          = [1.000000, 10.000000, 20.000000, 50.000000, 100.
000000]
RESOLUTIONS    = 0.150000
RANGES         = [-4912.000000,4912.000000]
DRI            = false
FIFO_SIZE      = 0
STREAM_TYPE    = streaming
STREAM_SYNC    = false
DYNAMIC        = false
EVENT_SIZE     = 16
OP_MODES       = [LOW_POWER, LOW_NOISE]
ACTIVE_CURRENT = 900
SLEEP_CURRENT  = 3
HW_ID          = 0
RIGID_BODY     = display
PHYSICAL_SENSOR       = true
PHYSICAL_SENSOR_TESTS = [3, 1]
SELECTED_RESOLUTION   = 0.150000
SELECTED_RANGE        = [-4912.000000,4912.000000]

SUID           = 0xaddeaddeaddeaddeaddeaddeaddeadde
NAME           = da_test
VENDOR         = QTI
TYPE           = da_test
AVAILABLE      = true
VERSION        = 0.51.0
API            = sns_da_test.proto
STREAM_TYPE    = on_change

SUID           = 0x69def905fea2fbac6a43ca273221a2eb
NAME           = aont
VENDOR         = QTI
TYPE           = aont
AVAILABLE      = true
VERSION        = 0.0.1
API            = sns_aont.proto
STREAM_TYPE    = on_change

SUID           = 0xabababababababababababababababab
NAME           = suid
VENDOR         = QTI
```

```
TYPE           = suid
AVAILABLE      = true
VERSION        = 0.0.1
API            = sns_suid.proto
STREAM_TYPE    = single_output

SUID           = 0xe12754a7007f27595e2541b4701e2275
NAME           = registry
VENDOR         = QTI
TYPE           = registry
AVAILABLE      = true
VERSION        = 0.0.1
API            = sns_registry.proto
STREAM_TYPE    = single_output

SUID           = 0xadfeadfeadfeadfeadfeadfeadfeadfe
NAME           = da_test_big_image
VENDOR         = QTI
TYPE           = da_test
AVAILABLE      = true
VERSION        = 0.49.0
API            = sns_da_test.proto
STREAM_TYPE    = on_change

SUID           = 0x61ab5376b4a5c9aa58442ede47acd316
NAME           = icm4x6xx
VENDOR         = TDK-Invensense
TYPE           = accel
AVAILABLE      = true
VERSION        = 1.65.2
API            = sns_accel.proto
RATES          = [12.500000, 25.000000, 50.000000, 100.000000, 200.
000000, 500.000000]
RESOLUTIONS    = [0.000019, 0.000037, 0.000075, 0.000150, 0.000299]
RANGES         = [[-9.806650,9.806650], [-19.613300,19.613300], [-39.
226601,39.226601], [-78.453201,78.453201], [-156.906403,156.906403]]
DRI            = true
FIFO_SIZE      = 200
STREAM_TYPE    = streaming
STREAM_SYNC    = false
DYNAMIC        = false
EVENT_SIZE     = 16
OP_MODES       = [LPM, NORMAL, HIGH_PERF]
ACTIVE_CURRENT = [50, 240, 240]
```

```
SLEEP_CURRENT  = 6
HW_ID          = 0
RIGID_BODY     = display
PHYSICAL_SENSOR       = true
PHYSICAL_SENSOR_TESTS = [3, 2, 1]
SELECTED_RESOLUTION   = 0.000299
```

For troubleshooting common issues, see Troubleshoot sensors.

The following QXDM output shows the verification/confirmation logs for all the supported sensors and their attributes in a low-power processor:

Note: This feature is currently available only for Authorized users. To upgrade your access, go to: www.qualcomm.com/support/working-with-qualcomm.

```
[ 123/   2]   MSG    18:47:22.898750     SNS PLATFORM/High
[sns_registry_parser.c   2099] REG INIT DONE... ts=63248955
[ 122/   1]   MSG    18:47:22.925000     SNS FRAMEWORK/Medium       [
 sns_diag_service.c   1072] Sensor registry Vendor:QTI SSID:53
registered with diag
[ 122/   1]   MSG    18:47:22.925000     SNS FRAMEWORK/Medium       [
       sns_sensor.c    769] Populating sensor b2926130 in island: 2,
with SUID e12754a7 007f2759 5e2541b4 701e2275
[ 122/   2]   MSG    18:47:22.925000     SNS FRAMEWORK/High
[sns_attribute_service.c    584] Sensor : registry, suid_populated :
1, available : 1
[ 122/   2]   MSG    18:47:22.925000     SNS FRAMEWORK/High
[sns_attribute_service.c    594] Avail:    B2926130  registry
[ 123/   2]   MSG    18:47:22.925000     SNS PLATFORM/High
[sns_registry_sensor.c    284] Successfully initialized registry
[ 122/   1]   MSG    18:47:22.925000     SNS FRAMEWORK/Medium
[sns_stream_service.c    150] Created data stream to Sensor 'registry
' (b2926130) from Sensor 'suid' (b2922018): b296fd28
[ 122/   1]   MSG    18:47:22.925000     SNS FRAMEWORK/Medium
[sns_stream_service.c    686] Add request b296fdc0 on stream b296fd28
(length 20; ID 512)
[ 122/   0]   MSG    18:47:22.926250     SNS FRAMEWORK/Low
[sns_stream_service.c    548] Process request for Sensor b2926130 on
b296fd28
[ 122/   0]   MSG    18:47:22.926250     SNS FRAMEWORK/Low          [
  sns_suid_sensor.c   1050] send_suid_event dt=registry, default=1
[ 123/   2]   MSG    18:47:22.926250     SNS PLATFORM/High          [
        sns_q6_pm.c    365] Deregister_client: B2927B30
```

```
[ 123/   2]   MSG    18:47:22.926250     SNS PLATFORM/High
[sns_registry_sensor.c    366] Processed : SSCRPCD UP Signal
[ 122/   0]   MSG    18:47:22.926250     SNS FRAMEWORK/Low          [
  sns_suid_sensor.c   1050] send_suid_event dt=registry, default=1
[ 122/   1]   MSG    18:47:22.926250     SNS FRAMEWORK/Medium       [
  sns_data_stream.c    551] sns_data_stream_deinit b29231d0
removing=0
[ 122/   1]   MSG    18:47:22.926250     SNS FRAMEWORK/Medium
[sns_stream_service.c    686] Add request b2927b30 on stream b29231d0
(length 0; ID 120)
[ 122/   0]   MSG    18:47:22.926250     SNS FRAMEWORK/Low          [
  sns_suid_sensor.c   1050] send_suid_event dt=registry, default=1
[ 123/   2]   MSG    18:47:22.926250     SNS PLATFORM/High          [
  sns_gdsc_island.c    109] gdsc_client_cnt(2), sns_gdsc_mode(1)
[ 122/   1]   MSG    18:47:22.926250     SNS FRAMEWORK/Medium
[sns_stream_service.c    150] Created data stream to Sensor 'timer'
(b2118e08) from Sensor 'power_sensor' (b2922c78): b2931c48
[ 122/   0]   MSG    18:47:22.926250     SNS FRAMEWORK/Low          [
  sns_suid_sensor.c   1050] send_suid_event dt=registry, default=1
[ 122/   1]   MSG    18:47:22.926250     SNS FRAMEWORK/Medium
[sns_stream_service.c    686] Add request b2123580 on stream b2931c48
(length 16; ID 512)
[ 122/   0]   MSG    18:47:22.926250     SNS FRAMEWORK/Low
[sns_stream_service.c    548] Process request for Sensor b2118e08 on
b2931c48
[ 122/   0]   MSG    18:47:22.926250     SNS FRAMEWORK/Low          [
  sns_suid_sensor.c   1050] send_suid_event dt=registry, default=1
```

For more information about an individual sensor level verification, see Tools.

4.3 Test sensors on platform

To use a sensor test application, see Test sensors on platform.

4.4 Troubleshoot sensors

The following resources describe a few common issues and the techniques available to analyze and troubleshoot these issues.

Note: The <registry_path> should be considered as one of the existing paths on the device: `/etc/sensors/registry/registry/` or

/var/cache/sensors/registry/registry/.

The `ssc_sensor_info` tool is unable to list sensors

This error indicates a failure in parsing the sensor registry, rendering the sensor unavailable for listing using the ssc_sensor_info tool. Do the following to resolve this error:

Note: Verify that the read and write permissions for the files are enabled for User mode, and read only permission is enabled for Group and Others modes. Ensure that the User and Group mode names are system system.

1. Ensure that /etc/sensors/sns_reg_config and /etc/sensors/config/json.lst files are present with the required permissions as shown in the following example:

```
root@qcm6490:~# ls -l /etc/sensors/sns_reg_config
-rw-r--r--. 1 system system 226 Mar  9  2018 /etc/
sensors/sns_reg_config
```

```
root@qcm6490:~# ls -l /etc/sensors/config/json.lst
-rw-r--r--. 1 system system 1452 Mar 9  2018 /etc/
sensors/config/json.lst
```

2. Ensure that /etc/sensors/config/ directory has the sensor JSON file with the required permissions.

```
root@qcm6490:/etc/sensors/config# ls -l
total 548
-rw-r--r--. 1 system system 1452 Mar 9  2018 json.lst
-rw-r--r--. 1 system system 5638 Mar 9  2018 kodiak_
ak991x_0.json
-rw-r--r--. 1 system system 2837 Mar 9  2018 kodiak_
bu52053nvx_0.json
-rw-r--r--. 1 system system 5480 Mar 9  2018 kodiak_
default_sensors.json
-rw-r--r--. 1 system system  293 Mar 9  2018 kodiak_
dynamic_sensors.json
-rw-r--r--. 1 system system  392 Mar 9  2018 kodiak_
idp_ak991x_0.json
-rw-r--r--. 1 system system  384 Mar 9  2018 kodiak_
idp_lsm6dst_0.json
-rw-r--r--. 1 system system  384 Mar 9  2018 kodiak_
```

```
idp_lsm6dst_1.json
..
-rw-r--r--. 1 system system  302 Mar 9  2018 qcm6490_
power_0.json
-rw-r--r--. 1 system system 7952 Mar 9  2018 qcm6490_
rbx_icm4x6xx_0.json
-rw-r--r--. 1 system system 5922 Mar 9  2018 qcm6490_
rbx_navmez_ak991x_0.json
-rw-r--r--. 1 system system 7957 Mar 9  2018 qcm6490_
rbx_navmez_icm4x6xx_0.json
-rw-r--r--. 1 system system 7957 Mar 9  2018 qcm6490_
rbx_navmez_rev2_icm4x6xx_0.json
-rw-r--r--. 1 system system 3688 Mar 9  2018 qcm6490_
rbx_navmezz_icp101xx_0.json
```

3. Ensure that `<registry_path>` directory is accessible and has the corresponding parsed file.

```
root@qcm6490:/var/cache/sensors/registry/registry# ls
-l
total 524
-rw-r--r--. 1 system system    3 Jan  1 00:00 DIR
-rw-r--r--. 1 system system  902 Apr 28  2022
qcm6490_default_sensors.json.default_sensors
-rw-r--r--. 1 system system   86 Apr 28  2022
qcm6490_default_sensors.json.default_sensors.accel
-rw-r--r--. 1 system system  133 Apr 28  2022
qcm6490_default_sensors.json.default_sensors.accel.
attr_0
-rw-r--r--. 1 system system   90 Apr 28  2022
qcm6490_default_sensors.json.default_sensors.accel_
cal
..
-rw-r--r--. 1 system system   84 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0.temp
-rw-r--r--. 1 system system  232 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0.temp.
config
-rw-r--r--. 1 system system  346 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_
platform
-rw-r--r--. 1 system system   95 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_
platform.accel
```

```
-rw-r--r--. 1 system system  146 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_
platform.accel.fac_cal
-rw-r--r--. 1 system system  183 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_
platform.accel.fac_cal.bias
-rw-r--r--. 1 system system  445 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_
platform.accel.fac_cal.corr_mat
-rw-r--r--. 1 system system  863 Apr 28  2022
qcm6490_rbx_navmez_icm4x6xx_0.json.icm4x6xx_0_
platform.config
```

SUID lookup failures

This error indicates a custom code failure in retrieving SUID for a specified sensor. Do the following to resolve this error:

1. Run the `ssc_sensor_info` tool to check if the specified sensor is available.

   - If the sensor list is empty, see the first debugging method described in The ssc_sensor_info tool is unable to list sensors.

   - If only the specified sensor is unavailable, proceed with the following checks:

     a. Ensure that the sensor is listed in `/etc/sensors/config/json.lst` file, as shown in the following example:

     ```
     root@qcm6490:~# cat /etc/sensors/config/
     json.lst
     kodiak_ak991x_0.json
     qcm6490_default_sensors.json
     qcm6490_power_0.json
     qcm6490_rbx_icm4x6xx_0.json
     qcm6490_rbx_navmez_icm4x6xx_0.json
     qcm6490_rbx_navmezz_icp101xx_0.json
     qcm6490_rbx_navmez_ak991x_0.json
     ```

     b. Verify that the configuration file for the specified sensor is parsed and present in the `parsed_file_list.csv` file, as shown in the following example:

     ```
     root@qcm6490:/var/cache/sensors/registry#
     cat parsed_file_list.csv
     qcm6490_rbx_navmez_icm4x6xx_0.json.
     ```

```
icm4x6xx_0_platform.config
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.orient
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.gyro.fac_cal.corr_mat
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.gyro.fac_cal.bias
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.gyro.fac_cal
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.gyro
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel.fac_cal.corr_
mat
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel.fac_cal.bias
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel.fac_cal
qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel
```

c. Verify that the configuration file for the specified sensor is parsed and present in the `<registry_path>` directory as shown in the following example:

```
root@qcm6490:/var/cache/sensors/registry/
registry# ls -l
total 524
-rw-r--r--. 1 system system  248 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0
-rw-r--r--. 1 system system   85 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.accel
-rw-r--r--. 1 system system  233 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.accel.config
-rw-r--r--. 1 system system   88 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.freefall
-rw-r--r--. 1 system system  236 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.freefall.config
-rw-r--r--. 1 system system   84 Apr 28
```

```
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.gyro
-rw-r--r--. 1 system system  232 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.gyro.config
-rw-r--r--. 1 system system   82 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.md
-rw-r--r--. 1 system system  230 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.md.config
-rw-r--r--. 1 system system   84 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.temp
-rw-r--r--. 1 system system  232 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0.temp.config
-rw-r--r--. 1 system system  346 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform
-rw-r--r--. 1 system system   95 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel
-rw-r--r--. 1 system system  146 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel.fac_cal
-rw-r--r--. 1 system system  183 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel.fac_cal.bias
-rw-r--r--. 1 system system  445 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.accel.fac_cal.corr_
mat
-rw-r--r--. 1 system system  863 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.config
-rw-r--r--. 1 system system   94 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.gyro
-rw-r--r--. 1 system system  145 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.gyro.fac_cal
-rw-r--r--. 1 system system  182 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
```

```
icm4x6xx_0_platform.gyro.fac_cal.bias
-rw-r--r--. 1 system system  444 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.gyro.fac_cal.corr_mat
-rw-r--r--. 1 system system   91 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.md
-rw-r--r--. 1 system system  192 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.md.config
-rw-r--r--. 1 system system  174 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.orient
-rw-r--r--. 1 system system  536 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.placement
-rw-r--r--. 1 system system   94 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.temp
-rw-r--r--. 1 system system  142 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.temp.fac_cal
-rw-r--r--. 1 system system  102 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.temp.fac_cal.bias
-rw-r--r--. 1 system system  103 Apr 28
2022 qcm6490_rbx_navmez_icm4x6xx_0.json.
icm4x6xx_0_platform.temp.fac_cal.scale
```

2. If the preceding validations didn’t result in any anomalies, then there could be an issue with your own Apps side custom code implementation that uses the QSH client APIs.

Sensor is listed but unable to receive sensor data

This error indicates failure of the sample or custom code, use the following sensor tool to receive events for a specified sensor.

Run the following command to stream the specified sensor:

`see_workhorse [-sensor=][-sample_rate=] [-batch_period=] [-calibrated=<0 | 1>] [-wakeup=<0 | 1>]` For example, `see_workhorse -sensor=accel -sample_rate=max -duration=30 -display_events=1`

If the above tool is able to receive the sensor event then there could be an issue with your Apps side custom code implementation that uses the QSH client APIs.

For more information, see Tools.

5 References

5.1 Related documents

List of resources to develop your own drivers.

These resources are available only for the authorized users. To upgrade your access, go to https://www.qualcomm.com/support.

Title	Number
Qualcomm Technologies, Inc.	
Qualcomm Linux Build Guide	80-70018-254
Qualcomm Linux Yocto Guide	80-70018-27
RB3 Gen 2 Development Kit	80-70018-251

5.2 Acronyms and terms

Sensors-related acronyms and expansions for better understanding.

Acronym or term	Definition
ALS	Ambient light sensor
ASCP	Asynchronous COM port
DRI	Data ready interrupt
EIS	Electronic image stabilization
GRV	Game rotation vector
HLOS	High level operating system
I^2C	Inter-integrated circuit
I^3C	Improved I^2C
IBI	In-band interrupt
IMU	Inertial measurement unit
MCPS	Million cycles per second
MTU	Maximum transmission unit
ODR	Output data rate

Acronym or term	Definition
PD	Protection domain
QSH	Qualcomm sensing hub
QuP	Qualcomm universal peripherals
QuRT	Qualcomm real time operating system
RTOS	Real time operating system
SCP	Synchronous COM port
SCons	Software construction
SMP2P	Shared memory point to point
SoC	System-on-chip
SPI	Serial peripheral interface
SSC	Qualcomm® Snapdragon™ sensors core
SUID	Sensor unique identifier
UART	Universal asynchronous receiver/transmitter

LEGAL INFORMATION

Your access to and use of this material, along with any documents, software, specifications, reference board files, drawings, diagnostics and other information contained herein (collectively this "Material"), is subject to your (including the corporation or other legal entity you represent, collectively "You" or "Your") acceptance of the terms and conditions ("Terms of Use") set forth below. If You do not agree to these Terms of Use, you may not use this Material and shall immediately destroy any copy thereof.

1) Legal Notice.

This Material is being made available to You solely for Your internal use with those products and service offerings of Qualcomm Technologies, Inc. ("**Qualcomm Technologies**"), its affiliates and/or licensors described in this Material, and shall not be used for any other purposes. If this Material is marked as "**Qualcomm Internal Use Only**", no license is granted to You herein, and You must immediately (a) destroy or return this Material to Qualcomm Technologies, and (b) report Your receipt of this Material to qualcomm.support@qti.qualcomm.com. This Material may not be altered, edited, or modified in any way without Qualcomm Technologies' prior written approval, nor may it be used for any machine learning or artificial intelligence development purpose which results, whether directly or indirectly, in the creation or development of an automated device, program, tool, algorithm, process, methodology, product and/or other output. Unauthorized use or disclosure of this Material or the information contained herein is strictly prohibited, and You agree to indemnify Qualcomm Technologies, its affiliates and licensors for any damages or losses suffered by Qualcomm Technologies, its affiliates and/or licensors for any such unauthorized uses or disclosures of this Material, in whole or part.

Qualcomm Technologies, its affiliates and/or licensors retain all rights and ownership in and to this Material. No license to any trademark, patent, copyright, mask work protection right or any other intellectual property right is either granted or implied by this Material or any information disclosed herein, including, but not limited to, any license to make, use, import or sell any product, service or technology offering embodying any of the information in this Material.

THIS MATERIAL IS BEING PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY LAW, QUALCOMM TECHNOLOGIES, ITS AFFILIATES AND/OR LICENSORS SPECIFICALLY DISCLAIM ALL WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, COMPLETENESS OR ACCURACY, AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MOREOVER, NEITHER QUALCOMM TECHNOLOGIES, NOR ANY OF ITS AFFILIATES AND/OR LICENSORS, SHALL BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY EXPENSES, LOSSES, USE, OR ACTIONS HOWSOEVER INCURRED OR UNDERTAKEN BY YOU IN RELIANCE ON THIS MATERIAL.

Certain product kits, tools and other items referenced in this Material may require You to accept additional terms and conditions before accessing or using those items.

Technical data specified in this Material may be subject to U.S. and other applicable export control laws. Transmission contrary to U.S. and any other applicable law is strictly prohibited.

Nothing in this Material is an offer to sell any of the components or devices referenced herein.

This Material is subject to change without further notification.

In the event of a conflict between these Terms of Use and the *Website Terms of Use* on www.qualcomm.com, the *Qualcomm Privacy Policy* referenced on www.qualcomm.com, or other legal statements or notices found on prior pages of the Material, these Terms of Use will control. In the event of a conflict between these Terms of Use and any other agreement (written or click-through, including, without limitation any non-disclosure agreement) executed by You and Qualcomm Technologies or a Qualcomm Technologies affiliate and/or licensor with respect to Your access to and use of this Material, the other agreement will control.

These Terms of Use shall be governed by and construed and enforced in accordance with the laws of the State of California, excluding the U.N. Convention on International Sale of Goods, without regard to conflict of laws principles. Any dispute, claim or controversy arising out of or relating to these Terms of Use, or the breach or validity hereof, shall be adjudicated only by a court of competent jurisdiction in the county of San Diego, State of California, and You hereby consent to the personal jurisdiction of such courts for that purpose.

2) Trademark and Product Attribution Statements.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. The Bluetooth® word mark is a registered trademark owned by Bluetooth SIG, Inc. Other product and brand names referenced in this Material may be trademarks or registered trademarks of their respective owners.

Snapdragon and Qualcomm branded products referenced in this Material are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.